

cloudera[®]

Apache Kudu Guide

Important Notice

© 2010-2021 Cloudera, Inc. All rights reserved.

Cloudera, the Cloudera logo, and any other product or service names or slogans contained in this document are trademarks of Cloudera and its suppliers or licensors, and may not be copied, imitated or used, in whole or in part, without the prior written permission of Cloudera or the applicable trademark holder. If this documentation includes code, including but not limited to, code examples, Cloudera makes this available to you under the terms of the Apache License, Version 2.0, including any required notices. A copy of the Apache License Version 2.0, including any notices, is included herein. A copy of the Apache License Version 2.0 can also be found here: <https://opensource.org/licenses/Apache-2.0>

Hadoop and the Hadoop elephant logo are trademarks of the Apache Software Foundation. All other trademarks, registered trademarks, product names and company names or logos mentioned in this document are the property of their respective owners. Reference to any products, services, processes or other information, by trade name, trademark, manufacturer, supplier or otherwise does not constitute or imply endorsement, sponsorship or recommendation thereof by us.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Cloudera.

Cloudera may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Cloudera, the furnishing of this document does not give you any license to these patents, trademarks copyrights, or other intellectual property. For information about patents covering Cloudera products, see <http://tiny.cloudera.com/patents>.

The information in this document is subject to change without notice. Cloudera shall not be liable for any damages resulting from technical errors or omissions which may be present in this document, or from use of this document.

Cloudera, Inc.

**395 Page Mill Road
Palo Alto, CA 94306
info@cloudera.com
US: 1-888-789-1488
Intl: 1-650-362-0488
www.cloudera.com**

Release Information

Version: Kudu 1.10.0 / CDH 6.3.x
Date: September 30, 2021

Table of Contents

Apache Kudu Overview.....	9
Kudu-Impala Integration.....	9
Example Use Cases.....	10
Related Information.....	10
Apache Kudu Concepts and Architecture.....	11
Columnar Datastore.....	11
Raft Consensus Algorithm.....	11
Table.....	11
Tablet.....	11
Tablet Server.....	11
Master.....	12
Catalog Table.....	12
Logical Replication.....	12
Architectural Overview.....	12
Apache Kudu Requirements.....	14
Apache Kudu Usage Limitations.....	15
Schema Design Limitations.....	15
Partitioning Limitations.....	16
Scaling Recommendations and Limitations.....	16
Server Management Limitations.....	16
Cluster Management Limitations.....	17
Replication and Backup Limitations.....	17
Impala Integration Limitations.....	17
Spark Integration Limitations.....	18
Security Limitations.....	18
Other Known Issues.....	18
Apache Kudu Installation and Upgrade.....	19
Platform Requirements.....	19
Downloading and Installing Kudu.....	19

Upgrading Kudu.....	21
Apache Kudu Configuration.....	22
Directory Configurations.....	22
Configuring the Kudu Master.....	23
Configuring Tablet Servers.....	23
Configuring Kudu Tables.....	24
Apache Kudu Administration.....	25
Starting and Stopping Kudu Processes.....	25
Kudu Web Interfaces.....	25
<i>Kudu Master Web Interface.....</i>	<i>25</i>
<i>Kudu Tablet Server Web Interface.....</i>	<i>25</i>
<i>Common Web Interface Pages.....</i>	<i>25</i>
Kudu Metrics.....	26
<i>Listing Available Metrics.....</i>	<i>26</i>
<i>Collecting Metrics via HTTP.....</i>	<i>26</i>
<i>Diagnostics Logging.....</i>	<i>27</i>
Rack Awareness (Location Awareness).....	28
Backup and Restore.....	28
<i>Logical backup and restore.....</i>	<i>28</i>
<i>Physical backups of an entire node.....</i>	<i>30</i>
Common Kudu Workflows.....	31
<i>Migrating to Multiple Kudu Masters.....</i>	<i>31</i>
<i>Recovering from a Dead Kudu Master in a Multi-Master Deployment.....</i>	<i>35</i>
<i>Removing Kudu Masters from a Multi-Master Deployment.....</i>	<i>38</i>
<i>Changing Master Hostnames.....</i>	<i>39</i>
<i>Best Practices when Adding New Tablet Servers.....</i>	<i>40</i>
<i>Monitoring Cluster Health with ksck.....</i>	<i>40</i>
<i>Changing Directory Configuration.....</i>	<i>42</i>
<i>Recovering from Disk Failure.....</i>	<i>42</i>
<i>Recovering from Full Disks.....</i>	<i>43</i>
<i>Bringing a Tablet That Has Lost a Majority of Replicas Back Online.....</i>	<i>44</i>
<i>Rebuilding a Kudu Filesystem Layout.....</i>	<i>45</i>
<i>Scaling Storage on Kudu Master and Tablet Servers in the Cloud.....</i>	<i>45</i>
Migrating Kudu Data from One Directory to Another on the Same Host.....	46
Minimizing Cluster Disruption During Temporary Planned Downtime of a Single Tablet Server.....	46
Running Tablet Rebalancing Tool.....	47
<i>Running Tablet Rebalancing Tool on Rack-Aware Cluster.....</i>	<i>48</i>
<i>Running Tablet Rebalancing Tool in Cloudera Manager.....</i>	<i>49</i>
Decommissioning or Permanently Removing a Tablet Server From a Cluster.....	49
Using cluster names in the <code>kudu</code> command line tool.....	49

Managing Kudu Using Cloudera Manager.....51

Installing and Upgrading the Kudu Service.....51
Enabling Core Dump for the Kudu Service.....51
Verifying the Impala Dependency on Kudu.....51
Using the Charts Library with the Kudu Service.....51

Developing Applications With Apache Kudu.....53

Viewing the API Documentation.....53
Kudu Example Applications.....53
Maven Artifacts.....54
Building the Java Client.....54
Kudu Python Client.....54
Example Apache Impala Commands With Kudu.....56
Kudu Integration with Spark.....56
Spark Integration Best Practices.....58
Integration with MapReduce, YARN, and Other Frameworks.....59

Using Apache Impala with Kudu.....60

Impala Database Containment Model.....60
Internal and External Impala Tables.....60
Using Impala To Query Kudu Tables.....61
Querying an Existing Kudu Table from Impala.....61
Creating a New Kudu Table From Impala.....61
Partitioning Tables.....62
Optimizing Performance for Evaluating SQL Predicates.....66
Inserting a Row.....66
Updating a Row.....67
Upserting a Row.....67
Altering a Table.....68
Deleting a Row.....69
Failures During INSERT, UPDATE, UPSERT, and DELETE Operations.....69
Altering Table Properties.....69
Dropping a Kudu Table using Impala.....70
Security Considerations.....70
Known Issues and Limitations.....70
Next Steps.....71

Using the Hive Metastore with Kudu.....72

Databases and Table Names.....72

<i>Databases</i>	72
<i>Naming Constraints</i>	72
<i>Metadata Synchronization</i>	72
Enabling the Hive Metastore Integration.....	73
Administrative Tools.....	74
Upgrading Existing Tables.....	75
<i>Prepare for the Upgrade</i>	75
<i>Perform the Upgrade</i>	75

Kudu Security.....77

Kudu Authentication with Kerberos.....	77
<i>Internal Private Key Infrastructure (PKI)</i>	77
<i>Authentication Tokens</i>	77
<i>Client Authentication to Secure Kudu Clusters</i>	78
Scalability.....	78
Coarse-grained Authorization.....	78
Fine-Grained Authorization.....	78
<i>Apache Sentry</i>	78
<i>Authorization Tokens</i>	79
<i>Trusted Users</i>	80
<i>Configuring the Integration with Apache Sentry</i>	80
<i>Caching</i>	80
<i>Policy for Kudu Masters</i>	80
<i>Policy for Kudu Tablet Servers</i>	81
Encryption.....	82
Web UI Encryption.....	82
Web UI Redaction.....	82
Log Redaction.....	82
Configuring a Secure Kudu Cluster using Cloudera Manager.....	83
Configuring a Secure Kudu Cluster using the Command Line.....	85

Apache Kudu Schema Design.....86

The Perfect Schema.....	86
Column Design.....	86
<i>Decimal Type</i>	87
<i>Column Encoding</i>	87
<i>Column Compression</i>	88
Primary Key Design.....	88
<i>Primary Key Index</i>	88
<i>Considerations for Backfill Inserts</i>	88
Partitioning.....	89
<i>Range Partitioning</i>	89

<i>Hash Partitioning</i>	90
<i>Multilevel Partitioning</i>	90
<i>Partition Pruning</i>	90
<i>Partitioning Examples</i>	90
Schema Alterations.....	93
Schema Design Limitations.....	93

Apache Kudu Transaction Semantics.....94

Single Tablet Write Operations.....	94
Writing to Multiple Tablets.....	94
Read Operations (Scans).....	95
Known Issues and Limitations.....	96
<i>Writes</i>	96
<i>Reads (Scans)</i>	97

Apache Kudu Background Maintenance Tasks.....98

Kudu Scaling Guide.....100

Terms.....	100
Example Workload.....	100
Memory.....	100
<i>Verifying if a Memory Limit is sufficient</i>	101
File Descriptors.....	102
Threads.....	102

Troubleshooting Apache Kudu.....104

Issues Starting or Restarting the Master or Tablet Server.....	104
<i>Errors During Hole Punching Test</i>	104
<i>Already present: FS layout already exists</i>	104
<i>NTP Clock Synchronization</i>	105
Disk Space Usage.....	107
Reporting Kudu Crashes Using Breakpad.....	108
Troubleshooting Performance Issues.....	109
<i>Kudu Tracing</i>	109
<i>Memory Limits</i>	110
<i>Block Cache Size</i>	111
<i>Heap Sampling</i>	112
<i>Slow Name Resolution and nscd</i>	112
Usability Issues.....	112
<i>ClassNotFoundException: com.cloudera.kudu.hive.KuduStorageHandler</i>	112

Runtime Error: Could not create thread: Resource temporarily unavailable (error 11).....112
Tombstoned or STOPPED tablet replicas.....113
Corruption: checksum error on CFile block.....113
Generating a Table List.....113
Spark Tuning.....114

More Resources for Apache Kudu.....115

Cloudera Manager Metrics for Kudu.....116

Appendix: Apache License, Version 2.0.....117

Apache Kudu Overview

Apache Kudu is a columnar storage manager developed for the Hadoop platform. Kudu shares the common technical properties of Hadoop ecosystem applications: It runs on commodity hardware, is horizontally scalable, and supports highly available operation.

Apache Kudu is a top-level project in the Apache Software Foundation.

Kudu's benefits include:

- Fast processing of OLAP workloads.
- Integration with MapReduce, Spark, Flume, and other Hadoop ecosystem components.
- Tight integration with Apache Impala, making it a good, mutable alternative to using HDFS with Apache Parquet.
- Strong but flexible consistency model, allowing you to choose consistency requirements on a per-request basis, including the option for strict serialized consistency.
- Strong performance for running sequential and random workloads simultaneously.
- Easy administration and management through Cloudera Manager.
- High availability. Tablet Servers and Master use the Raft consensus algorithm, which ensures availability as long as more replicas are available than unavailable. Reads can be serviced by read-only follower tablets, even in the event of a leader tablet failure.
- Structured data model.

By combining all of these properties, Kudu targets support applications that are difficult or impossible to implement on currently available Hadoop storage technologies. Applications for which Kudu is a viable solution include:

- Reporting applications where new data must be immediately available for end users
- Time-series applications that must support queries across large amounts of historic data while simultaneously returning granular queries about an individual entity
- Applications that use predictive models to make real-time decisions, with periodic refreshes of the predictive model based on all historical data

Kudu-Impala Integration

Apache Kudu has tight integration with Apache Impala, allowing you to use Impala to insert, query, update, and delete data from Kudu tablets using Impala's SQL syntax, as an alternative to using the Kudu APIs to build a custom Kudu application. In addition, you can use JDBC or ODBC to connect existing or new applications written in any language, framework, or business intelligence tool to your Kudu data, using Impala as the broker.

- **CREATE/ALTER/DROP TABLE** - Impala supports creating, altering, and dropping tables using Kudu as the persistence layer. The tables follow the same internal/external approach as other tables in Impala, allowing for flexible data ingestion and querying.
- **INSERT** - Data can be inserted into Kudu tables from Impala using the same mechanisms as any other table with HDFS or HBase persistence.
- **UPDATE/DELETE** - Impala supports the `UPDATE` and `DELETE` SQL commands to modify existing data in a Kudu table row-by-row or as a batch. The syntax of the SQL commands is designed to be as compatible as possible with existing solutions. In addition to simple `DELETE` or `UPDATE` commands, you can specify complex joins in the `FROM` clause of the query, using the same syntax as a regular `SELECT` statement.
- **Flexible Partitioning** - Similar to partitioning of tables in Hive, Kudu allows you to dynamically pre-split tables by hash or range into a predefined number of tablets, in order to distribute writes and queries evenly across your cluster. You can partition by any number of primary key columns, with any number of hashes, a list of split rows, or a combination of these. A partition scheme is required.
- **Parallel Scan** - To achieve the highest possible performance on modern hardware, the Kudu client used by Impala parallelizes scans across multiple tablets.

- **High-efficiency queries** - Where possible, Impala pushes down predicate evaluation to Kudu, so that predicates are evaluated as close as possible to the data. Query performance is comparable to Parquet in many workloads.

Example Use Cases

Streaming Input with Near Real Time Availability

A common business challenge is one where new data arrives rapidly and constantly, and the same data needs to be available in near real time for reads, scans, and updates. Kudu offers the powerful combination of fast inserts and updates with efficient columnar scans to enable real-time analytics use cases on a single storage layer.

Time-Series Application with Widely Varying Access Patterns

A time-series schema is one in which data points are organized and keyed according to the time at which they occurred. This can be useful for investigating the performance of metrics over time or attempting to predict future behavior based on past data. For instance, time-series customer data might be used both to store purchase click-stream history and to predict future purchases, or for use by a customer support representative. While these different types of analysis are occurring, inserts and mutations might also be occurring individually and in bulk, and become available immediately to read workloads. Kudu can handle all of these access patterns simultaneously in a scalable and efficient manner.

Kudu is a good fit for time-series workloads for several reasons. With Kudu's support for hash-based partitioning, combined with its native support for compound row keys, it is simple to set up a table spread across many servers without the risk of "hotspotting" that is commonly observed when range partitioning is used. Kudu's columnar storage engine is also beneficial in this context, because many time-series workloads read only a few columns, as opposed to the whole row.

In the past, you might have needed to use multiple datastores to handle different data access patterns. This practice adds complexity to your application and operations, and duplicates your data, doubling (or worse) the amount of storage required. Kudu can handle all of these access patterns natively and efficiently, without the need to off-load work to other datastores.

Predictive Modeling

Data scientists often develop predictive learning models from large sets of data. The model and the data might need to be updated or modified often as the learning takes place or as the situation being modeled changes. In addition, the scientist might want to change one or more factors in the model to see what happens over time. Updating a large set of data stored in files in HDFS is resource-intensive, as each file needs to be completely rewritten. In Kudu, updates happen in near real time. The scientist can tweak the value, re-run the query, and refresh the graph in seconds or minutes, rather than hours or days. In addition, batch or incremental algorithms can be run across the data at any time, with near-real-time results.

Combining Data In Kudu With Legacy Systems

Companies generate data from multiple sources and store it in a variety of systems and formats. For instance, some of your data might be stored in Kudu, some in a traditional RDBMS, and some in files in HDFS. You can access and query all of these sources and formats using Impala, without the need to change your legacy systems.

Related Information

- [Apache Kudu Concepts and Architecture](#) on page 11
- [Apache Kudu Installation and Upgrade](#) on page 19
- [Kudu Security](#) on page 77
- [More Resources for Apache Kudu](#) on page 115

Apache Kudu Concepts and Architecture

Columnar Datastore

Kudu is a *columnar datastore*. A columnar datastore stores data in strongly-typed columns. With a proper design, a columnar store can be superior for analytical or data warehousing workloads for the following reasons:

Read Efficiency

For analytical queries, you can read a single column, or a portion of that column, while ignoring other columns. This means you can fulfill your request while reading a minimal number of blocks on disk. With a row-based store, you need to read the entire row, even if you only return values from a few columns.

Data Compression

Because a given column contains only one type of data, pattern-based compression can be orders of magnitude more efficient than compressing mixed data types, which are used in row-based solutions. Combined with the efficiencies of reading data from columns, compression allows you to fulfill your query while reading even fewer blocks from disk.

Raft Consensus Algorithm

The [Raft consensus algorithm](#) provides a way to elect a *leader* for a distributed cluster from a pool of potential leaders. If a follower cannot reach the current leader, it transitions itself to become a *candidate*. Given a quorum of voters, one candidate is elected to be the new leader, and the others transition back to being followers. A full discussion of Raft is out of scope for this documentation, but it is a robust algorithm.

Kudu uses the Raft Consensus Algorithm for the election of masters and leader tablets, as well as determining the success or failure of a given write operation.

Table

A *table* is where your data is stored in Kudu. A table has a schema and a totally ordered primary key. A table is split into segments called tablets, by primary key.

Tablet

A *tablet* is a contiguous segment of a table, similar to a *partition* in other data storage engines or relational databases. A given tablet is replicated on multiple tablet servers, and at any given point in time, one of these replicas is considered the leader tablet. Any replica can service reads. Writes require consensus among the set of tablet servers serving the tablet.

Tablet Server

A *tablet server* stores and serves tablets to clients. For a given tablet, one tablet server acts as a leader and the others serve follower replicas of that tablet. Only leaders service write requests, while leaders or followers each service read requests. Leaders are elected using Raft consensus. One tablet server can serve multiple tablets, and one tablet can be served by multiple tablet servers.

Master

The *master* keeps track of all the tablets, tablet servers, the catalog table, and other metadata related to the cluster. At a given point in time, there can only be one acting master (the leader). If the current leader disappears, a new master is elected using Raft consensus.

The master also coordinates metadata operations for clients. For example, when creating a new table, the client internally sends the request to the master. The master writes the metadata for the new table into the catalog table, and coordinates the process of creating tablets on the tablet servers.

All the master's data is stored in a tablet, which can be replicated to all the other candidate masters.

Tablet servers heartbeat to the master at a set interval (the default is once per second).

Catalog Table

The *catalog table* is the central location for metadata of Kudu. It stores information about tables and tablets. The catalog table is accessible to clients through the master, using the client API. The catalog table cannot be read or written directly. Instead, it is accessible only through metadata operations exposed in the client API. The catalog table stores two categories of metadata:

Contents of the Catalog Table	
Tables	Table schemas, locations, and states
Tablets	The list of existing tablets, which tablet servers have replicas of each tablet, the tablet's current state, and start and end keys.

Logical Replication

Kudu replicates operations, not on-disk data. This is referred to as *logical replication*, as opposed to *physical replication*. This has several advantages:

- Although inserts and updates transmit data over the network, deletes do not need to move any data. The delete operation is sent to each tablet server, which performs the delete locally.
- Physical operations, such as compaction, do not need to transmit the data over the network in Kudu. This is different from storage systems that use HDFS, where the blocks need to be transmitted over the network to fulfill the required number of replicas.
- Tablets do not need to perform compactions at the same time or on the same schedule. They do not even need to remain in sync on the physical storage layer. This decreases the chances of all tablet servers experiencing high latency at the same time, due to compactions or heavy write loads.

Architectural Overview

The following diagram shows a Kudu cluster with three masters and multiple tablet servers, each serving multiple tablets. It illustrates how Raft consensus is used to allow for both leaders and followers for both the masters and tablet servers. In addition, a tablet server can be a leader for some tablets and a follower for others. Leaders are shown in gold, while followers are shown in grey.

Kudu network architecture

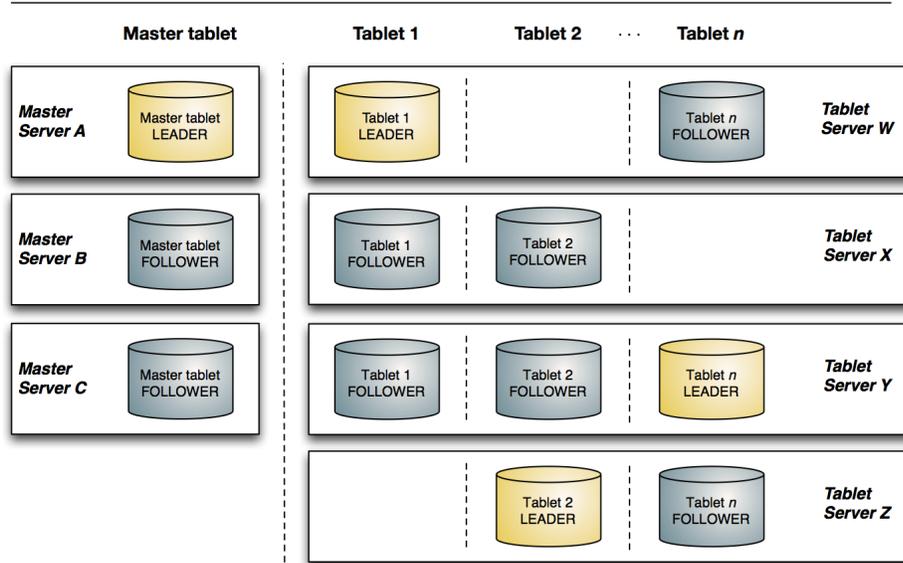


Figure 1: Kudu Architectural Overview

Apache Kudu Requirements

Starting with Kudu 1.5.0 / CDH 5.13, Kudu is fully integrated in the CDH 5 parcel and packages. As such, for the complete list of hardware and software requirements for Kudu, see the [Product Compatibility Matrix for Apache Kudu](#).

Apache Kudu Usage Limitations

Schema Design Limitations

Primary Key

- The primary key cannot be changed after the table is created. You must drop and recreate a table to select a new primary key.
- The columns which make up the primary key must be listed first in the schema.
- The primary key of a row cannot be modified using the `UPDATE` functionality. To modify a row's primary key, the row must be deleted and re-inserted with the modified key. Such a modification is non-atomic.
- Columns with `DOUBLE`, `FLOAT`, or `BOOL` types are not allowed as part of a primary key definition. Additionally, all columns that are part of a primary key definition must be `NOT NULL`.
- Auto-generated primary keys are not supported.
- Cells making up a composite primary key are limited to a total of 16KB after internal composite-key encoding is done by Kudu.

Cells

No individual cell may be larger than 64KB before encoding or compression. The cells making up a composite key are limited to a total of 16KB after the internal composite-key encoding done by Kudu. Inserting rows not conforming to these limitations will result in errors being returned to the client.

Columns

- By default, Kudu tables can have a maximum of 300 columns. We recommend schema designs that use fewer columns for best performance.
- `CHAR`, `VARCHAR`, `DATE`, and complex types such as `ARRAY`, `MAP`, and `STRUCT` are not supported.
- Type and nullability of existing columns cannot be changed by altering the table.
- Dropping a column does not immediately reclaim space. Compaction must run first.
- The precision and scale of `DECIMAL` columns cannot be changed by altering the table.

Tables

- Tables must have an odd number of replicas, with a maximum of 7.
- Replication factor (set at table creation time) cannot be changed.
- There is no way to run compaction manually, but dropping a table will reclaim the space immediately.

Other Usage Limitations

- Secondary indexes are not supported.
- Multi-row transactions are not supported.
- Relational features, such as foreign keys, are not supported.
- Identifiers such as column and table names are restricted to be valid UTF-8 strings. Additionally, a maximum length of 256 characters is enforced.

If you are using Apache Impala to query Kudu tables, refer to the section on [Impala Integration Limitations](#) on page 17 as well.

Partitioning Limitations

- Tables must be manually pre-split into tablets using simple or compound primary keys. Automatic splitting is not yet possible. Kudu does not allow you to change how a table is partitioned after creation, with the exception of adding or dropping range partitions.
- Data in existing tables cannot currently be automatically repartitioned. As a workaround, create a new table with the new partitioning and insert the contents of the old table.
- Tablets that lose a majority of replicas (such as 1 left out of 3) require manual intervention to be repaired.

Scaling Recommendations and Limitations

Kudu can seamlessly run across a wide array of environments and workloads with minimal expertise and configuration at the following scale:

- Recommended maximum number of masters is 3.
- Recommended maximum number of tablet servers is 100.
- Recommended maximum amount of stored data, post-replication and post-compression, per tablet server is 8 TiB.
- Recommended number of tablets per tablet server is 1000 (post-replication) with 2000 being the maximum number of tablets allowed per tablet server.
- Maximum number of tablets per table is 60, per tablet server, at table-creation time.
- Maximum number of tablets per table for each tablet server is 60, post-replication (assuming the default replication factor of 3), at table-creation time.
- Recommended maximum amount of data per tablet is 50 GiB. Going beyond this can cause issues such as a reduced performance, compaction issues, and slow tablet startup times.

The recommended target size for tablets is under 10 GiB.

Staying within these limits will provide the most predictable and straightforward Kudu experience. However, experienced users who run on modern hardware, use the latest versions of Kudu, test and tune Kudu for their use case, and work closely with the community, can achieve much higher scales comfortably. Following are some anecdotal values that have been seen in real world production clusters:

- Number of master servers: 3
- More than 300 tablet servers
- 10+ TiB of stored data per tablet server, post-replication and post-compression
- More than 4000 tablets per tablet server, post-replication
- 50 GiB of stored data per tablet. Going beyond this can cause issues such as a reduced performance, compaction issues, and slower tablet startup time.

Server Management Limitations

- Production deployments should configure a least 4 GiB of memory for tablet servers, and ideally more than 16 GiB when approaching the data and tablet scale limits.
- Write ahead logs (WALs) can only be stored on one disk.

- Data directories cannot be removed. You must reformat the data directories to remove them.
- Tablet servers cannot be gracefully decommissioned.
- Tablet servers cannot change their address or port.
- Kudu has a hard requirement on having an up-to-date NTP. Kudu masters and tablet servers will crash when out of sync.
- Kudu releases have only been tested with NTP. Other time synchronization providers such as Chrony may not work.

Cluster Management Limitations

- Rolling restart is not supported.
- Recommended maximum point-to-point latency within a Kudu cluster is 20 milliseconds.
- Recommended minimum point-to-point bandwidth within a Kudu cluster is 10 Gbps.
- If you intend to use the location awareness feature to place tablet servers in different locations, it is recommended that you measure the bandwidth and latency between servers to ensure they fit within the above guidelines.
- All masters must be started at the same time when the cluster is started for the very first time.

Replication and Backup Limitations

- Kudu does not currently include any built-in features for backup and restore. Users are encouraged to use tools such as Spark or Impala to export or import tables as necessary.

Impala Integration Limitations

- When creating a Kudu table, the `CREATE TABLE` statement must include the primary key columns before other columns, in primary key order.
- Impala cannot update values in primary key columns.
- Impala cannot create Kudu tables with `VARCHAR` or nested-typed columns.
- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when used as an external table in Impala.
- Kudu tables with a column name containing upper case or non-ASCII characters cannot be used as an external table in Impala. Columns can be renamed in Kudu to work around this issue.
- `!=` and `LIKE` predicates are not pushed to Kudu, and instead will be evaluated by the Impala scan node. This may decrease performance relative to other types of predicates.
- Updates, inserts, and deletes using Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or use large tables.

Impala Keywords Not Supported for Creating Kudu Tables

- `PARTITIONED`

Apache Kudu Usage Limitations

- LOCATION
- ROWFORMAT

Spark Integration Limitations

- Spark 2.2 (and higher) requires Java 8 at runtime even though Kudu Spark 2.x integration is Java 7 compatible. Spark 2.2 is the default dependency version as of Kudu 1.5.0.
- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when registered as a temporary table.
- Kudu tables with a column name containing upper case or non-ASCII characters must not be used with SparkSQL. Columns can be renamed in Kudu to work around this issue.
- `<>` and `OR` predicates are not pushed to Kudu, and instead will be evaluated by the Spark task. Only `LIKE` predicates with a suffix wildcard are pushed to Kudu. This means `LIKE "FOO%"` will be pushed, but `LIKE "FOO%BAR"` won't.
- Kudu does not support all the types supported by Spark SQL. For example, `Date` and complex types are not supported.
- Kudu tables can only be registered as temporary tables in SparkSQL.
- Kudu tables cannot be queried using `HiveContext`.

Security Limitations

- Data encryption at rest is not directly built into Kudu. Encryption of Kudu data at rest can be achieved through the use of local block device encryption software such as `dmccrypt`.
- Row-level authorization is not available.
- Kudu does not support configuring a custom service principal for Kudu processes. The principal must follow the pattern `kudu/<HOST>@<DEFAULT.REALM>`.
- Server certificates generated by Kudu IPKI are incompatible with [bouncycastle](#) version 1.52 and earlier.
- The highest supported version of the TLS protocol is TLSv1.2

Other Known Issues

The following are known bugs and issues with the current release of Kudu. They will be addressed in later releases. Note that this list is not exhaustive, and is meant to communicate only the most important known issues.

- If the Kudu master is configured with the `-log_force_fsync_all` option, the tablet servers and the clients will experience frequent timeouts, and the cluster may become unusable.
- If a tablet server has a very large number of tablets, it may take several minutes to start up. It is recommended to limit the number of tablets per server to 1000 or fewer. Consider this limitation when pre-splitting your tables. If you notice slow start-up times, you can monitor the number of tablets per server in the web UI.

Apache Kudu Installation and Upgrade

Starting with Apache Kudu 1.5.0 / CDH 5.13, Kudu ships with CDH 5. In a parcel-based configuration, Kudu is part of the CDH parcel rather than a separate parcel. The Kudu packages are also bundled into the CDH package.

Platform Requirements

Before you proceed with installation or upgrade, review [Platform Requirements](#).

Downloading and Installing Kudu

Minimum Required Role: [Cluster Administrator](#)

On a cluster managed by Cloudera Manager, Kudu is installed as part of CDH and does not need to be installed separately. With Cloudera Manager, you can enable or disable the Kudu service, but the Kudu component remains present on the cluster. For instructions, see [Cloudera Installation Guide](#).

Starting CDH 6.3.3, you must have an Enterprise Support Subscription to access `archive.cloudera.com`.

If you want support for Kudu's NVM (non-volatile memory) block cache, then you must install the `memkind` library by running one of the following commands for the corresponding platform:

- **RHEL/CentOS:**

```
sudo yum install memkind
```

- **Ubuntu or Debian:**

```
sudo apt-get install libmemkind0
```

- **SLES:**

```
sudo zypper install memkind
```



Note: For Kudu's NVM block cache to work properly, the `memkind` library should be of version 1.8.0 or newer.

If the `memkind` package provided with the Linux distribution is too old, you can build and install it from source by using the following commands:

- **RHEL/CentOS:**

```
sudo yum install numactl-libs numactl-devel
```

```
git clone https://github.com/memkind/memkind.git
```

```
cd memkind
```

```
./build.sh --prefix=/usr
```

```
sudo yum remove memkind
```

```
sudo make install
```

```
sudo ldconfig
```

- **Ubuntu or Debian:**

```
sudo apt-get install libnuma1 libnuma-dev
```

```
git clone https://github.com/memkind/memkind.git
```

```
cd memkind
```

```
./build.sh --prefix=/usr
```

```
sudo apt-get remove memkind
```

```
sudo make install
```

```
sudo ldconfig
```

- **SLES:**

```
sudo zypper install numactl-libs numactl-devel
```

```
git clone https://github.com/memkind/memkind.git
```

```
cd memkind
```

```
./build.sh --prefix=/usr
```

```
sudo zypper remove memkind
```

```
sudo make install
```

```
sudo ldconfig
```

Upgrading Kudu

Before you proceed with an upgrade, review the [Upgrade Notes](#)

On a managed cluster,

- If you have just upgraded Cloudera Manager from a version that did not include Kudu, then Kudu will not be installed automatically. You will need to add the Kudu service manually. Upgrading Cloudera Manager does not automatically upgrade CDH or other managed services.
- **Parcels:** If you are upgrading CDH and were previously using the standalone Kudu parcel (version 1.4.0 and lower), then you must deactivate this parcel and activate the latest CDH parcel that includes Kudu. For instructions, see [Upgrade Guide](#)
- **Packages:** If you are upgrading CDH and were previously using the Kudu package (version 1.4.0 and lower), then you must uninstall the `kudu` package and upgrade to the latest CDH package that includes Kudu. For instructions, see [Upgrade Guide](#)

Apache Kudu Configuration

To configure the behavior of each Kudu process, you can pass command-line flags when you start it, or read those options from configuration files by passing them using one or more `--flagfile=<file>` options. You can even include the `--flagfile` option within your configuration file to include other files. Learn more about gflags by reading [its documentation](#).

You can place options for masters and tablet servers in the same configuration file, and each will ignore options that do not apply.

Flags can be prefixed with either one or two `-` characters. This documentation standardizes on two: `--example_flag`.

Only the most common configuration options are documented in this topic. For a more exhaustive list of configuration options, see the [Kudu Configuration Reference](#). To see all configuration flags for a given executable, run it with the `--help` option.

Experimental Flags

Some configuration flags are marked 'unsafe' and 'experimental'. Such flags are disabled by default. You can access these flags by enabling the additional flags, `--unlock_unsafe_flags` and `--unlock_experimental_flags`. Note that these flags might be removed or modified without a deprecation period or any prior notice in future Kudu releases. Cloudera does not support using unsafe and experimental flags. As a rule of thumb, Cloudera will not support any configuration flags not explicitly documented in the [Kudu Configuration Reference Guide](#).

Directory Configurations

Every Kudu node requires the specification of directory flags.

The `--fs_wal_dir` configuration indicates where Kudu will place its write-ahead logs.

The `--fs_metadata_dir` configuration indicates where Kudu will place metadata for each tablet. It is recommended, although not necessary, that these directories be placed on a high-performance drives with high bandwidth and low latency, e.g. solid-state drives. If `--fs_metadata_dir` is not specified, metadata will be placed in the directory specified by `--fs_wal_dir`.

Since a Kudu node cannot tolerate the loss of its WAL or metadata directories, you might want to mirror the drives containing these directories in order to make recovering from a drive failure easier. However, mirroring may increase the latency of Kudu writes.

The `--fs_data_dirs` configuration indicates where Kudu will write its data blocks. This is a comma-separated list of directories; if multiple values are specified, data will be striped across the directories. If not specified, data blocks will be placed in the directory specified by `--fs_wal_dir`.



Note: While a single data directory backed by a RAID-0 array will outperform a single data directory backed by a single storage device, it is better to let Kudu manage its own striping over multiple devices rather than delegating the striping to a RAID-0 array.

Additionally, `--fs_wal_dir` and `--fs_metadata_dir` may be the same as *one of* the directories listed in `--fs_data_dirs`, but must not be sub-directories of any of them.



Warning: Each directory specified by a configuration flag on a given machine should be used by at most one Kudu process. If multiple Kudu processes on the same machine are configured to use the same directory, Kudu may refuse to start up.



Warning: Once `--fs_data_dirs` is set, extra tooling is required to change it. For more details, see the [Changing Directory Configuration](#) on page 42.



Note: The `--fs_wal_dir` and `--fs_metadata_dir` configurations can be changed, provided the contents of the directories are also moved to match the flags.

Configuring the Kudu Master

To see all available configuration options for the `kudu-master` executable, run it with the `--help` option:

```
kudu-master --help
```

Table 1: Supported Configuration Flags for Kudu Masters

Flag	Valid Options	Default	Description
<code>--master_addresses</code>	string	localhost	Comma-separated list of all the RPC addresses for Master consensus-configuration. If not specified, assumes a standalone Master.
<code>--fs_data_dirs</code>	string		List of directories where the Master will place its data blocks.
<code>--fs_wal_dir</code>	string		The directory where the Master will place its write-ahead logs.
<code>--log_dir</code>	string	/tmp	The directory to store Master log files.

For the complete list of flags for masters, see the [Kudu Master Configuration Reference](#).

Configuring Tablet Servers

To see all available configuration options for the `kudu-tserver` executable, run it with the `--help` option:

```
kudu-tserver --help
```

Table 2: Supported Configuration Flags for Kudu Tablet Servers

Flag	Valid Options	Default	Description
<code>--fs_data_dirs</code>	string		List of directories where the Tablet Server will place its data blocks.

Flag	Valid Options	Default	Description
--fs_wal_dir	string		The directory where the Tablet Server will place its write-ahead logs.
--log_dir	string	/tmp	The directory to store Tablet Server log files
--tserver_master_addrs	string	127.0.0.1:7051	Comma separated addresses of the masters that the tablet server should connect to. The masters do not read this flag.
--block_cache_capacity_mb	integer	512	Maximum amount of memory allocated to the Kudu Tablet Server's block cache.
--memory_limit_hard_bytes	integer	4294967296	Maximum amount of memory a Tablet Server can consume before it starts rejecting all incoming writes.

For the complete list of flags for tablet servers, see the [Kudu Tablet Server Configuration Reference](#).

Configuring Kudu Tables

Kudu allows certain configurations to be set per table. To configure the behavior of a Kudu table, you can set these configurations at table creation, or alter them via the Kudu API or Kudu command line tool.

Table 3: Supported Configurable Properties for Kudu Tables

Configuration	Valid Options	Default	Description
kudu.table.history_max_age_sec	Integer	NA	Number of seconds to retain history for tablets in this table.

Apache Kudu Administration

This topic describes how to perform common administrative tasks and workflows with Apache Kudu.

Starting and Stopping Kudu Processes

Start Kudu services using the following commands:

```
sudo service kudu-master start
sudo service kudu-tserver start
```

To stop Kudu services, use the following commands:

```
sudo service kudu-master stop
sudo service kudu-tserver stop
```

Configure the Kudu services to start automatically when the server starts, by adding them to the default runlevel.

```
sudo chkconfig kudu-master on           # RHEL / CentOS
sudo chkconfig kudu-tserver on         # RHEL / CentOS

sudo update-rc.d kudu-master defaults  # Ubuntu
sudo update-rc.d kudu-tserver defaults # Ubuntu
```

Kudu Web Interfaces

Kudu tablet servers and masters expose useful operational information on a built-in web interface.

Kudu Master Web Interface

Kudu master processes serve their web interface on port 8051. The interface exposes several pages with information about the state of the cluster.

- A list of tablet servers, their host names, and the time of their last heartbeat.
- A list of tables, including schema and tablet location information for each.
- SQL code which you can paste into Impala Shell to add an existing table to Impala's list of known data sources.

Kudu Tablet Server Web Interface

Each tablet server serves a web interface on port 8050. The interface exposes information about each tablet hosted on the server, its current state, and debugging information about maintenance background operations.

Common Web Interface Pages

Both Kudu masters and tablet servers expose the following information via their web interfaces:

- HTTP access to server logs.
- An `/rpcz` endpoint which lists currently running RPCs via JSON.
- Details about the memory usage of different components of the process.
- The current set of configuration flags.
- Currently running threads and their resource consumption.

- A JSON endpoint exposing metrics about the server.
- The version number of the daemon deployed on the cluster.

These interfaces are linked from the landing page of each daemon's web UI.

Kudu Metrics

Kudu daemons expose a large number of metrics. Some metrics are associated with an entire server process, whereas others are associated with a particular tablet replica.

Listing Available Metrics

The full set of available metrics for a Kudu server can be dumped using a special command line flag:

```
$ kudu-tserver --dump_metrics_json  
$ kudu-master --dump_metrics_json
```

This will output a large JSON document. Each metric indicates its name, label, description, units, and type. Because the output is JSON-formatted, this information can easily be parsed and fed into other tooling which collects metrics from Kudu servers.

For the complete list of metrics collected by Cloudera Manager for a Kudu service, look for the Kudu metrics listed under [Cloudera Manager Metrics](#).

If you are using Cloudera Manager, see [Cloudera Manager Metrics for Kudu](#) on page 116 for the complete list of metrics collected by Cloudera Manager for a Kudu service.

Collecting Metrics via HTTP

Metrics can be collected from a server process via its HTTP interface by visiting `/metrics`. The output of this page is JSON for easy parsing by monitoring services. This endpoint accepts several `GET` parameters in its query string:

- `/metrics?metrics=<substring1>,<substring2>,...` - Limits the returned metrics to those which contain at least one of the provided substrings. The substrings also match entity names, so this may be used to collect metrics for a specific tablet.
- `/metrics?include_schema=1` - Includes metrics schema information such as unit, description, and label in the JSON output. This information is typically omitted to save space.
- `/metrics?compact=1` - Eliminates unnecessary whitespace from the resulting JSON, which can decrease bandwidth when fetching this page from a remote host.
- `/metrics?include_raw_histograms=1` - Include the raw buckets and values for histogram metrics, enabling accurate aggregation of percentile metrics over time and across hosts.
- `/metrics?level=info` - Limits the returned metrics based on their severity level. The levels are ordered and lower levels include the levels above them. If no level is specified, `debug` is used to include all metrics. The valid values are:
 - `debug` - Metrics that are diagnostically helpful but generally not monitored during normal operation.
 - `info` - Generally useful metrics that operators always want to have available but may not be monitored under normal circumstances.
 - `warn` - Metrics which can often indicate operational oddities, which may need more investigation.

For example:

```
$ curl -s 'http://example-ts:8050/metrics?include_schema=1&metrics=connections_accepted'
```

```
[
  {
    "type": "server",
    "id": "kudu.tabletserver",
    "attributes": {},
    "metrics": [
      {
        "name": "rpc_connections_accepted",
        "label": "RPC Connections Accepted",
        "type": "counter",
        "unit": "connections",
        "description": "Number of incoming TCP connections made to the RPC
server",
        "value": 92
      }
    ]
  }
]
```

```
$ curl -s 'http://example-ts:8050/metrics?metrics=log_append_latency'
```

```
[
  {
    "type": "tablet",
    "id": "c0ebf9fef1b847e2a83c7bd35c2056b1",
    "attributes": {
      "table_name": "lineitem",
      "partition": "hash buckets: (55), range: [(<start>), (<end>))",
      "table_id": ""
    },
    "metrics": [
      {
        "name": "log_append_latency",
        "total_count": 7498,
        "min": 4,
        "mean": 69.3649,
        "percentile_75": 29,
        "percentile_95": 38,
        "percentile_99": 45,
        "percentile_99_9": 95,
        "percentile_99_99": 167,
        "max": 367244,
        "total_sum": 520098
      }
    ]
  }
]
```

Diagnostics Logging

Kudu may be configured to periodically dump all of its metrics to a local log file using the `--metrics_log_interval_msflag`. Set this flag to the interval at which metrics should be written to a `diagnostics` log file.

The diagnostics log will be written to the same directory as the other Kudu log files, with a similar naming format, substituting `diagnostics` instead of a log level like `INFO`. After any diagnostics log file reaches 64MB uncompressed, the log will be rolled and the previous file will be gzip-compressed.

The log file generated has three space-separated fields. The first field is the word `metrics`. The second field is the current timestamp in microseconds since the Unix epoch. The third is the current value of all metrics on the server, using a compact JSON encoding. The encoding is the same as the metrics fetched via HTTP described above.

Rack Awareness (Location Awareness)

Starting in CDH 6.2, Kudu supports a rack awareness feature. Kudu's ordinary re-replication methods ensure the availability of the cluster in the event of a single node failure. However, clusters can be vulnerable to correlated failures of multiple nodes. For example, all of the physical hosts on the same rack in a datacenter may become unavailable simultaneously if the top-of-rack switch fails. Kudu's rack awareness feature provides protection from certain kinds of correlated failures, such as the failure of a single rack in a datacenter.

The first element of Kudu's rack awareness feature is location assignment. When a tablet server registers with a master, the master assigns it a location. A location is a `/`-separated string that begins with a `/` and where each `/`-separated component consists of characters from the set `[a-zA-Z0-9_-]`. For example, `/dc-0/rack-09` is a valid location, while `rack-04` and `/rack=1` are not valid locations. Thus location strings resemble absolute UNIX file paths where characters in directory and file names are restricted to the set `[a-zA-Z0-9_-]`. Presently, Kudu does not use the hierarchical structure of locations, but it may in the future. Location assignment is done by a user-provided command, whose path should be specified using the `--location_mapping_cmd` master flag. The command should take a single argument, the IP address or hostname of a tablet server, and return the location for the tablet server. Make sure that all Kudu masters are using the same location mapping command.

The second element of Kudu's rack awareness feature is the placement policy: Do not place a majority of replicas of a tablet on tablet servers in the same location.

The leader master, when placing newly created replicas on tablet servers and when re-replicating existing tablets, will attempt to place the replicas in a way that complies with the placement policy. For example, in a cluster with five tablet servers `A`, `B`, `C`, `D`, and `E`, with respective locations `/L0`, `/L0`, `/L1`, `/L1`, `/L2`, to comply with the placement policy a new 3x replicated tablet could have its replicas placed on `A`, `C`, and `E`, but not on `A`, `B`, and `C`, because then the tablet would have 2/3 replicas in location `/L0`. As another example, if a tablet has replicas on tablet servers `A`, `C`, and `E`, and then `C` fails, the replacement replica must be placed on `D` in order to comply with the placement policy.

In the case where it is impossible to place replicas in a way that complies with the placement policy, Kudu will violate the policy and place a replica anyway. For example, using the setup described in the previous paragraph, if a tablet has replicas on tablet servers `A`, `C`, and `E`, and then `E` fails, Kudu will re-replicate the tablet onto one of `B` or `D`, violating the placement policy, rather than leaving the tablet under-replicated indefinitely. The `kudu cluster rebalance` tool can reestablish the placement policy if it is possible to do so. The `kudu cluster rebalance` tool can also be used to reimpose the placement policy on a cluster if the cluster has just been configured to use the rack awareness feature and existing replicas need to be moved to comply with the placement policy. See [Running Tablet Rebalancing Tool on Rack-Aware Cluster](#) on page 48 for more information.

Backup and Restore

Logical backup and restore

As of Kudu 1.10.0, Kudu supports both full and incremental table backups via a job implemented using Apache Spark. Additionally, it supports restoring tables from full and incremental backups via a restore job implemented using Apache Spark.

Kudu backup and restore jobs use Apache Spark. Therefore, ensure that you install Apache Spark in your environment. To download Apache Spark, see the [Apache Spark documentation](#). You can also review the [Submitting Spark Applications](#) topics.

Backing up tables

You can use the `KuduBackup` Spark job to backup one or more Kudu tables. When you first run the job for a table, a full backup is run. Additional runs will perform incremental backups which will only contain the rows that have changed since the initial full backup. A new set of full backups can be forced at anytime by passing the `--forceFull` flag to the backup job.

Following are some of the common flags that you can use while taking a backup:

- `--rootPath`: The root path is used to output backup data. It accepts any Spark-compatible path.
- `--kuduMasterAddresses`: Is used to specify comma-separated addresses of Kudu masters. The default value is `localhost`.
- `<table>...`: Used to indicate a list of tables that you want to back up.



Note: You can obtain a full list of Job options by entering the `--help` flag.

Here is a full example of a `KuduBackup` job execution which backs up the tables `foo` and `bar` to the HDFS directory `kudu-backups`:

```
spark-submit --class org.apache.kudu.backup.KuduBackup kudu-backup2_2.11-1.10.0.jar \
  --kuduMasterAddresses master1-host,master-2-host,master-3-host \
  --rootPath hdfs:///kudu-backups \
  foo bar
```

Restoring Tables from Backups

You can use the `KuduRestore` Spark job to restore one or more Kudu tables. For each backed up table, the `KuduRestore` job restores the full backup and each associated incremental backup until the full table state is restored. Restoring a complete series of full and incremental backups is possible because the backups are linked via the `from_ms` and `to_ms` fields in the backup metadata. By default, the restore job creates tables with the same name as the table that was backed up. If you want to side-load the tables without affecting the existing tables, you can specify the `--tableSuffix` parameter to append a suffix to each restored table.

Some of the common flags that you can use to restore tables are:

- `--rootPath`: It is the root path to the backup data. It accepts any Spark-compatible path.
- `--kuduMasterAddresses`: Is used to specify comma-separated addresses of Kudu masters. The default value is `localhost`.
- `--tableSuffix`: It is used to add a suffix to the restored table names. It can only be used when `createTables` is set to `true`.
- `--timestampMs`: It is a UNIX timestamp in milliseconds that defines the latest time to use when selecting restore candidates. Default: `System.currentTimeMillis()`
- `<table>...`: It is used to specify a list of tables to be backed up.



Note: You can obtain a full list of Job options by entering the `--help` flag.

Here is a full example of a `KuduRestore` job execution which restores the tables `foo` and `bar` from the HDFS directory `kudu-backups`:

```
spark-submit --class org.apache.kudu.backup.KuduRestore kudu-backup2_2.11-1.10.0.jar \
  --kuduMasterAddresses master1-host,master-2-host,master-3-host \
  --rootPath hdfs:///kudu-backups \
  foo bar
```

Backup Tools

An additional `backup-tools` jar is available to provide some backup exploration and garbage collection capabilities. This jar does not use Spark directly, but instead only requires the Hadoop classpath to run.

Commands:

- `list`: Lists the backups in the `rootPath`
- `clean`: Cleans up old backed up data in the `rootPath`



Note: You can obtain a full list of Job options by entering the `--help` flag.

Following is an example execution which prints the command options:

```
java -cp $(hadoop classpath):kudu-backup-tools-1.10.0.jar
org.apache.kudu.backup.KuduBackupCLI --help
```

Backup Directory Structure

The backup directory structure in the `rootPath` is considered an internal detail and could change in future versions of Kudu. Additionally, the format and content of the data and metadata files is meant for the backup and restore process only and could change in future versions of Kudu. That said, understanding the structure of the backup `rootPath` and how it is used can be useful when working with Kudu backups.

The backup directory structure in the `rootPath` is as follows:

```
/<rootPath>/<tableId>-<tableName>/<backup-id>/
.kudu-metadata.json
part-*.<format>
```

- `rootPath`: Can be used to distinguish separate backup groups, jobs, or concerns.
- `tableId`: Is the unique internal ID of the table being backed up.
- `tableName`: Is the name of the table being backed up.
Note: Table names are URL encoded to prevent pathing issues.
- `backup-id`: Is a way to uniquely identify/group the data for a single backup run.
- `.kudu-metadata.json`: It contains all of the metadata to support recreating the table, linking backups by time, and handling data format changes.

It is written last so that the failed backups will not have a metadata file and will not be considered at the restore time or at the backup linking time.

- `part-*.<format>`: Is used to indicate the data files containing the tables data.
 - Currently, 1 part file per Kudu partition.
 - Incremental backups contain an additional “RowAction” byte column at the end.
 - Currently, the only supported format/suffix is `parquet`

Physical backups of an entire node

Kudu does not provide a built-in physical backup and restore functionality yet. However, it is possible to create a physical backup of a Kudu node (either tablet server or master) and restore it later.



Warning:

The node to be backed up must be offline during the procedure, or else the backed up (or restored) data will be inconsistent.

Certain aspects of the Kudu node (such as its hostname) are embedded in the on-disk data. As such, it's not yet possible to restore a physical backup of a node onto another machine.

1. Stop all Kudu processes in the cluster. This prevents the tablets on the backed up node from being rereplicated elsewhere unnecessarily.
2. If creating a backup, make a copy of the WAL, metadata, and data directories on each node to be backed up. It is important that this copy preserve all file attributes as well as sparseness.
3. If restoring from a backup, delete the existing WAL, metadata, and data directories, then restore the backup via move or copy. As with creating a backup, it is important that the restore preserve all file attributes and sparseness.
4. Start all Kudu processes in the cluster.

Common Kudu Workflows

The following sections describe some common workflows for Kudu users:

Migrating to Multiple Kudu Masters

To provide high availability and to avoid a single point of failure, Kudu clusters should be created with multiple masters. Many Kudu clusters were created with just a single master, either for simplicity or because Kudu multi-master support was still experimental at the time. This workflow demonstrates how to migrate to a multi-master configuration. It can also be used to migrate from two masters to three with straightforward modifications.



Important:

- This workflow is unsafe for adding new masters to an existing multi-master configuration that already has three or more masters. Do not use it for that purpose.
- An even number of masters doesn't provide any benefit over having one fewer masters. This guide should always be used for migrating to three masters.
- This workflow presumes you are familiar with Kudu configuration management, with or without Cloudera Manager.
- All of the command line steps below should be executed as the Kudu UNIX user. The example commands assume the Kudu Unix user is `kudu`, which is typical.

Prepare for the migration

1. Establish a maintenance window (one hour should be sufficient). During this time the Kudu cluster will be unavailable.
2. Decide how many masters to use. The number of masters should be odd. Three or five node master configurations are recommended; they can tolerate one or two failures respectively.
3. Perform the following preparatory steps for the existing master:
 - Identify and record the directories where the master's write-ahead log (WAL) and data live. If using Kudu system packages, their default locations are `/var/lib/kudu/master`, but they may be customized using the `fs_wal_dir` and `fs_data_dirs` configuration parameters. The command below assume that `fs_wal_dir` is `/data/kudu/master/wal` and `fs_data_dirs` is `/data/kudu/master/data`. Your configuration may differ. For more information on configuring these directories, see the [Kudu Configuration docs](#).
 - Identify and record the port the master is using for RPCs. The default port value is 7051, but it may have been customized using the `rpc_bind_addresses` configuration parameter.
 - Identify the master's UUID. It can be fetched using the following command:

```
$ sudo -u kudu kudu fs dump uuid --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>] 2>/dev/null
```

master_data_dir

The location of the existing master's previously recorded data directory.

For example:

```
$ sudo -u kudu kudu fs dump uuid --fs_wal_dir=/var/lib/kudu/master 2>/dev/null
4aab798a69e94fab8d77069edff28ce0
```

- **(Optional)** Configure a DNS alias for the master. The alias could be a DNS cname (if the machine already has an A record in DNS), an A record (if the machine is only known by its IP address), or an alias in `/etc/hosts`. The alias should be an abstract representation of the master (e.g. `master-1`).



Important: Without DNS aliases, it is not possible to recover from permanent master failures without bringing the cluster down for maintenance. It is highly recommended that you use DNS aliases.

4. If you have Kudu tables that are accessed from Impala, you must update the master addresses in the Apache Hive Metastore (HMS) database.

- If you set up the DNS aliases, run the following statement in `impala-shell`, replacing `master-1`, `master-2`, and `master-3` with your actual aliases.

```
ALTER TABLE table_name
SET TBLPROPERTIES
('kudu.master_addresses' = 'master-1,master-2,master-3');
```

- If you do not have DNS aliases set up, see Step #11 in the Performing the migration section for updating HMS.

5. Perform the following preparatory steps for each new master:

- Choose an unused machine in the cluster. The master generates very little load so it can be collocated with other data services or load-generating processes, though not with another Kudu master from the same configuration.
- Ensure Kudu is installed on the machine, either using system packages (in which case the `kudu` and `kudu-master` packages should be installed), or some other means.
- Choose and record the directory where the master's data will live.
- Choose and record the port the master should use for RPCs.
- **(Optional)** Configure a DNS alias for the master (e.g. `master-2`, `master-3`, etc).

Perform the migration

1. Stop all the Kudu processes in the entire cluster.
2. Format the data directory on each new master machine, and record the generated UUID. Use the following commands:

```
$ sudo -u kudu kudu fs format --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>]
$ sudo -u kudu kudu fs dump uuid --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>] 2>/dev/null
```

master_data_dir

The new master's previously recorded data directory.

For example:

```
$ sudo -u kudu kudu fs format --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data
sudo -u kudu kudu fs dump uuid --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data 2>/dev/null
f5624e05f40649b79a757629a69d061e
```

3. If you are using Cloudera Manager, add the new Kudu master roles now, but do not start them.

- If using DNS aliases, override the empty value of the `Master Address` parameter for each role (including the existing master role) with that master's alias.
- Add the port number (separated by a colon) if using a non-default RPC port value.

4. Rewrite the master's Raft configuration with the following command, executed on the existing master:

```
$ sudo -u kudu kudu local_replica cmeta rewrite_raft_config --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>] <tablet_id> <all_masters>
```

master_data_dir

The existing master's previously recorded data directory

tablet_id

This must be set to the string, 00000000000000000000000000000000.

all_masters

A space-separated list of masters, both new and existing. Each entry in the list must be a string of the form `<uuid> : <hostname> : <port>`.

uuid

The master's previously recorded UUID.

hostname

The master's previously recorded hostname or alias.

port

The master's previously recorded RPC port number.

For example:

```
$ sudo -u kudu kudu local_replica cmeta rewrite_raft_config
--fs_wal_dir=/data/kudu/master/wal --fs_data_dirs=/data/kudu/master/data
00000000000000000000000000000000 4aab798a69e94fab8d77069edff28ce0:master-1:7051
f5624e05f40649b79a757629a69d061e:master-2:7051
988d8ac6530f426cbe180be5ba52033d:master-3:7051
```



Important: If you are using Cloudera Manager, skip the next step.

5. Modify the value of the `master_addresses` configuration parameter for both existing master and new masters. The new value must be a comma-separated list of all of the masters. Each entry is a string of the form, `<hostname> : <port>`.

hostname

The master's previously recorded hostname or alias.

port

The master's previously recorded RPC port number.

6. Start the existing master.
7. Copy the master data to each new master with the following command, executed on each new master machine.



Important: If your Kudu cluster is secure, in addition to running as the Kudu UNIX user, you must authenticate as the Kudu service user prior to running this command.

```
$ sudo -u kudu kudu local_replica copy_from_remote --fs_wal_dir=<master_data_dir>
<tablet_id> <existing_master>
```

master_data_dir

The new master's previously recorded data directory.

tablet_id

Must be set to the string, 00000000000000000000000000000000.

existing_master

RPC address of the existing master. It must be a string of the form <hostname>:<port>.

hostname

The existing master's previously recorded hostname or alias.

port

The existing master's previously recorded RPC port number.

Example

```
$ sudo -u kudu kudu local_replica copy_from_remote --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data 00000000000000000000000000000000 master-1:7051
```

8. Start all the new masters.



Important: If you are using Cloudera Manager, skip the next step.

9. Modify the value of the `tserver_master_addrs` configuration parameter for each tablet server. The new value must be a comma-separated list of masters where each entry is a string of the form <hostname>:<port>

hostname

The master's previously recorded hostname or alias

port

The master's previously recorded RPC port number

- 10 Start all the tablet servers.
- 11 If you have Kudu tables that are accessed from Impala and you didn't set up DNS aliases, update the HMS database manually in the underlying database that provides the storage for HMS.
 - The following is an example SQL statement you would run in the HMS database:

```
UPDATE TABLE_PARAMS
SET PARAM_VALUE =
'master-1.example.com,master-2.example.com,master-3.example.com'
WHERE PARAM_KEY = 'kudu.master_addresses' AND PARAM_VALUE = 'old-master';
```

- Invalidate the metadata by running the command in `impala-shell`:

```
INVALIDATE METADATA;
```

To verify that all masters are working properly, consider performing the following sanity checks:

- Using a browser, visit each master's web UI and navigate to the `/masters` page. All the masters should now be listed there with one master in the `LEADER` role and the others in the `FOLLOWER` role. The contents of `/masters` on each master should be the same.
- Run a Kudu system check (`ksck`) on the cluster using the `kudu` command line tool. For more details, see [Monitoring Cluster Health with `ksck`](#) on page 40.

Recovering from a Dead Kudu Master in a Multi-Master Deployment

Kudu multi-master deployments function normally in the event of a master loss. However, it is important to replace the dead master. Otherwise a second failure may lead to a loss of availability, depending on the number of available masters. This workflow describes how to replace the dead master.

Due to [KUDU-1620](#), it is not possible to perform this workflow without also restarting the live masters. As such, the workflow requires a maintenance window, albeit a potentially brief one if the cluster was set up with DNS aliases.



Important:

- Kudu does not yet support live Raft configuration changes for masters. As such, it is only possible to replace a master if the deployment was created with DNS aliases or if every node in the cluster is first shut down. See the previous [multi-master migration workflow](#) for more details on deploying with DNS aliases.
- The workflow presupposes at least basic familiarity with Kudu configuration management. If using Cloudera Manager, the workflow also presupposes familiarity with it.
- All of the command line steps below should be executed as the Kudu UNIX user, typically `kudu`.

Prepare for the recovery

1. If the cluster was configured without DNS aliases perform the following steps. Otherwise move on to step 2:
 - a. Establish a maintenance window (one hour should be sufficient). During this time the Kudu cluster will be unavailable.
 - b. Shut down all Kudu tablet server processes in the cluster.
2. Ensure that the dead master is well and truly dead. Take whatever steps needed to prevent it from accidentally restarting; this can be quite dangerous for the cluster post-recovery.
3. Choose one of the remaining live masters to serve as a basis for recovery. The rest of this workflow will refer to this master as the "reference" master.
4. Choose an unused machine in the cluster where the new master will live. The master generates very little load so it can be co-located with other data services or load-generating processes, though not with another Kudu master from the same configuration. The rest of this workflow will refer to this master as the "replacement" master.
5. Perform the following preparatory steps for the replacement master:
 - Ensure Kudu is installed on the machine, either via system packages (in which case the `kudu` and `kudu-master` packages should be installed), or via some other means.
 - Choose and record the directory where the master's data will live.
6. Perform the following preparatory steps for each live master:

- Identify and record the directory where the master’s data lives. If using Kudu system packages, the default value is `/var/lib/kudu/master`, but it may be customized via the `fs_wal_dir` and `fs_data_dirs` configuration parameter. Please note if you’ve set `fs_data_dirs` to some directories other than the value of `fs_wal_dir`, it should be explicitly included in every command below where `fs_wal_dir` is also included. For more information on configuring these directories, see the [Kudu Configuration docs](#).
- Identify and record the master’s UUID. It can be fetched using the following command:

```
$ sudo -u kudu kudu fs dump uuid --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>] 2>/dev/null
```

master_data_dir

live master’s previously recorded data directory

Example

```
$ sudo -u kudu kudu fs dump uuid --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data 2>/dev/null
80a82c4b8a9f4c819bab744927ad765c
```

7. Perform the following preparatory steps for the reference master:

- Identify and record the directory where the master’s data lives. If using Kudu system packages, the default value is `/var/lib/kudu/master`, but it may be customized using the `fs_wal_dir` and `fs_data_dirs` configuration parameter. If you have set `fs_data_dirs` to some directories other than the value of `fs_wal_dir`, it should be explicitly included in every command below where `fs_wal_dir` is also included. For more information on configuring these directories, see the [Kudu Configuration docs](#).
- Identify and record the UUIDs of every master in the cluster, using the following command:

```
$ sudo -u kudu kudu local_replica cmeta print_replica_uuids --fs_wal_dir=<master_data_dir>
<tablet_id> 2>/dev/null
```

master_data_dir

The reference master’s previously recorded data directory.

tablet_id

Must be set to the string, `00000000000000000000000000000000`.

For example

```
$ sudo -u kudu kudu local_replica cmeta print_replica_uuids
--fs_wal_dir=/data/kudu/master/wal --fs_data_dirs=/data/kudu/master/data
00000000000000000000000000000000 2>/dev/null
80a82c4b8a9f4c819bab744927ad765c 2a73eeee5d47413981d9a1c637cce170
1c3f3094256347528d02ec107466aef3
```

8. Using the two previously-recorded lists of UUIDs (one for all live masters and one for all masters), determine and record (by process of elimination) the UUID of the dead master.

Perform the recovery

1. Format the data directory on the replacement master machine using the previously recorded UUID of the dead master. Use the following command sequence:

```
$ sudo -u kudu kudu fs format --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>] --uuid=<uuid>
```

master_data_dir

The replacement master’s previously recorded data directory.

uuid

The dead master's previously recorded UUID.

For example:

```
$ sudo -u kudu kudu fs format --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data --uuid=80a82c4b8a9f4c819bab744927ad765c
```

2. Copy the master data to the replacement master with the following command.



Important: If your Kudu cluster is secure, in addition to running as the Kudu UNIX user, you must authenticate as the Kudu service user prior to running this command.

```
$ sudo -u kudu kudu local_replica copy_from_remote --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>] <tablet_id> <reference_master>
```

master_data_dir

The replacement master's previously recorded data directory.

tablet_id

Must be set to the string, 00000000000000000000000000000000.

reference_master

The RPC address of the reference master. It must be a string of the form <hostname>:<port>.

hostname

The reference master's previously recorded hostname or alias.

port

The reference master's previously recorded RPC port number.

For example:

```
$ sudo -u kudu kudu local_replica copy_from_remote --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data 00000000000000000000000000000000 master-2:7051
```

3. If you are using Cloudera Manager, add the replacement Kudu master role now, but do not start it.
 - Override the empty value of the `Master Address` parameter for the new role with the replacement master's alias.
 - If you are using a non-default RPC port, add the port number (separated by a colon) as well.
4. If the cluster was set up with DNS aliases, reconfigure the DNS alias for the dead master to point at the replacement master.
5. If the cluster was set up *without DNS aliases*, perform the following steps:
 - a. Stop the remaining live masters.
 - b. Rewrite the Raft configurations on these masters to include the replacement master. See Step 4 of [Perform the Migration](#) for more details.
6. Start the replacement master.
7. Restart the remaining masters in the new multi-master deployment. While the masters are shut down, there will be an availability outage, but it should last only as long as it takes for the masters to come back up.

To verify that all masters are working properly, consider performing the following sanity checks:

- Using a browser, visit each master's web UI and navigate to the `/masters` page. All the masters should now be listed there with one master in the `LEADER` role and the others in the `FOLLOWER` role. The contents of `/masters` on each master should be the same.
- Run a Kudu system check (`ksck`) on the cluster using the `kudu` command line tool. For more details, see [Monitoring Cluster Health with `ksck`](#) on page 40.

Removing Kudu Masters from a Multi-Master Deployment

In the event that a multi-master deployment has been overallocated nodes, the following steps should be taken to remove the unwanted masters.



Important:

- In planning the new multi-master configuration, keep in mind that the number of masters should be odd and that three or five node master configurations are recommended.
- Dropping the number of masters below the number of masters currently needed for a Raft majority can incur data loss. To mitigate this, ensure that the leader master is not removed during this process.

Prepare for removal

1. Establish a maintenance window (one hour should be sufficient). During this time the Kudu cluster will be unavailable.
2. Identify the UUID and RPC address current leader of the multi-master deployment by visiting the `/masters` page of any master's web UI. This master must not be removed during this process; its removal may result in severe data loss.
3. Stop all the Kudu processes in the entire cluster.
4. If you are using Cloudera Manager, remove the unwanted Kudu master from your cluster's Kudu service.

Perform the removal

1. Rewrite the Raft configuration on the remaining masters to include only the remaining masters. See Step 4 of [Perform the Migration](#) for more details.
2. Remove the data directories and WAL directory on the unwanted masters. This is a precaution to ensure that they cannot start up again and interfere with the new multi-master deployment.
3. Modify the value of the `master_addresses` configuration parameter for the masters of the new multi-master deployment. See [Kudu Configuration docs](#) for the steps to modify a configuration parameter. If migrating to a single-master deployment, the `master_addresses` flag should be omitted entirely.
4. Start all of the masters that were not removed.



Important: If you are using Cloudera Manager, skip the next step.

5. Modify the value of the `tserver_master_addr` configuration parameter for the tablet servers to remove any unwanted masters. See [Kudu Configuration docs](#) for the steps to modify a configuration parameter.
6. Start all of the tablet servers.

To verify that all masters are working properly, consider performing the following sanity checks:

- Using a browser, visit each master's web UI and navigate to the `/masters` page. All the masters should now be listed there with one master in the `LEADER` role and the others in the `FOLLOWER` role. The contents of `/masters` on each master should be the same.
- Run a Kudu system check (`ksck`) on the cluster using the `kudu` command line tool. For more details, see [Monitoring Cluster Health with `ksck`](#) on page 40.

Changing Master Hostnames

When replacing dead masters, use DNS aliases to prevent long maintenance windows. If the cluster was set up without aliases, change the host names as described in this section.

Prepare for Hostname Changes

To prepare to change a hostname:

1. Establish a maintenance window during which the Kudu cluster will be unavailable. One hour should be sufficient.
2. On the **Masters** page in Kudu Web UI, note the UUID and RPC address of each master.
3. Stop all the Kudu processes in the cluster.
4. Set up the new hostnames to point to the masters and verify all servers and clients properly resolve them.

Perform Hostname Changes

To change hostnames:

1. Rewrite each master's Raft configuration with the following command, executed on each master host:

```
$ sudo -u kudu kudu local_replica cmeta rewrite_raft_config --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>] 0000000000000000000000000000000000 <all_masters>
```

For example:

```
$ sudo -u kudu kudu local_replica cmeta rewrite_raft_config
--fs_wal_dir=/data/kudu/master/wal --fs_data_dirs=/data/kudu/master/data
0000000000000000000000000000000000 4aab798a69e94fab8d77069edff28ce0:new-master-name-1:7051
f5624e05f40649b79a757629a69d061e:new-master-name-2:7051
988d8ac6530f426cbe180be5ba52033d:new-master-name-3:7051
```

2. Update the master address:
 - In an environment not managed by Cloudera Manager, change the `gflag` file of the masters so the `master_addresses` parameter reflects the new hostnames.
 - In an environment managed by Cloudera Manager, specify the new hostname in the **Master Address (server.address)** field on each Kudu role.
3. Change the `gflag` file of the tablet servers to update the `tserver_master_addrs` parameter with the new hostnames. In an environment managed by Cloudera Manager, this step is not needed.
4. Start the masters.
5. To verify that all masters are working properly, perform the following sanity checks:
 - a. In each master's Web UI, click **Masters** on the Status Pages. All of the masters should be listed there with one master in the **LEADER** role field and the others in the **FOLLOWER** role field. The contents of **Masters** on all master should be the same.
 - b. Run the below command to verify all masters are up and listening. The UUIDs are the same and belong to the same master as before the hostname change:

```
$ sudo -u kudu kudu master list
new-master-name-1:7051,new-master-name-2:7051,new-master-name-3:7051
```

6. Start all of the tablet servers.
7. Run a Kudu system check (`ksck`) on the cluster using the `kudu` command line tool. See [Monitoring Cluster Health with ksck](#) on page 40 for more details. After startup, some tablets may be unavailable as it takes some time to initialize all of them.
8. If you have Kudu tables that are accessed from Impala, update the HMS database manually in the underlying database that provides the storage for HMS.

a. The following is an example SQL statement you run in the HMS database:

```
UPDATE TABLE_PARAMSSET PARAM_VALUE =
'new-master-name-1:7051,new-master-name-2:7051,new-master-name-3:7051'
WHERE PARAM_KEY = 'kudu.master_addresses'
AND PARAM_VALUE = 'master-1:7051, master-2:7051, master-3:7051';
```

b. In `impala-shell`, run:

```
INVALIDATE METADATA;
```

c. Verify updating the metadata worked by running a simple `SELECT` query on a Kudu-backed Impala table.

Best Practices when Adding New Tablet Servers

A common workflow when administering a Kudu cluster is adding additional tablet server instances, in an effort to increase storage capacity, decrease load or utilization on individual hosts, increase compute power, and more.

By default, any newly added tablet servers will not be utilized immediately after their addition to the cluster. Instead, newly added tablet servers will only be utilized when new tablets are created or when existing tablets need to be replicated, which can lead to imbalanced nodes. It's recommended to run the rebalancer CLI tool just after adding a new tablet server into the cluster.

Avoid placing multiple tablet servers on a single node. Doing so nullifies the point of increasing the overall storage capacity of a Kudu cluster and increases the likelihood of tablet unavailability when a single node fails (the latter drawback is not applicable if the cluster is properly configured to use the [Rack Awareness \(Location Awareness\)](#) on page 28 feature.

To add additional tablet servers to an existing cluster, the following steps can be taken to ensure tablet replicas are uniformly distributed across the cluster:

1. Ensure that Kudu is installed on the new machines being added to the cluster, and that the new instances have been correctly configured to point to the pre-existing cluster. Then, start the new tablet server instances.
2. Verify that the new instances check in with the Kudu Master(s) successfully. A quick method for verifying whether they have successfully checked in with the existing Master instances is to view the Kudu Master WebUI, specifically the `/tablet-servers` section, and validate that the newly added instances are registered, and have a heartbeat.
3. Once the tablet server(s) are successfully online and healthy, follow the steps to run the [rebalancing tool](#) which spreads the existing tablet replicas to the newly added tablet servers.
4. After the rebalancer tool has completed, or even during its execution, you can check the health of the cluster using the `ksck` command-line utility.

Monitoring Cluster Health with `ksck`

The `kudu` CLI includes a tool called `ksck` that can be used for gathering information about the state of a Kudu cluster, including checking its health. `ksck` will identify issues such as under-replicated tablets, unreachable tablet servers, or tablets without a leader.

`ksck` should be run from the command line as the Kudu admin user, and requires the full list of master addresses to be specified:

```
$ sudo -u kudu kudu cluster ksck
master-01.example.com,master-02.example.com,master-03.example.com
```

To see a full list of the options available with `ksck`, use the `--help` flag. If the cluster is healthy, `ksck` will print information about the cluster, a success message, and return a zero (success) exit status.

Master Summary		
UUID	Address	Status
a811c07b99394df799e6650e7310f282	master-01.example.com	HEALTHY
b579355eaaaa446e998606bcb7e87844	master-02.example.com	HEALTHY

```

cfdcc8592711485fad32ec4eea4fbfcd | master-02.example.com | HEALTHY

Tablet Server Summary
      UUID | Address | Status
-----+-----+-----
a598f75345834133a39c6e51163245db | tserver-01.example.com | HEALTHY
e05ca6b6573b4e1f9a518157c0c0c637 | tserver-02.example.com | HEALTHY
e7e53a91fe704296b3a59ad304e7444a | tserver-03.example.com | HEALTHY

Version Summary
Version | Servers
-----+-----
  1.7.1 | all 6 server(s) checked

Summary by table
Name | RF | Status | Total Tablets | Healthy | Recovering | Under-replicated |
Unavailable
-----+-----+-----+-----+-----+-----+-----
my_table | 3 | HEALTHY | 8 | 8 | 0 | 0

      | Total Count
-----+-----
Masters | 3
Tablet Servers | 3
Tables | 1
Tablets | 8
Replicas | 24
OK

```

If the cluster is unhealthy, for instance if a tablet server process has stopped, `ksck` will report the issue(s) and return a non-zero exit status, as shown in the abbreviated snippet of `ksck` output below:

```

Tablet Server Summary
      UUID | Address | Status
-----+-----+-----
a598f75345834133a39c6e51163245db | tserver-01.example.com | HEALTHY
e05ca6b6573b4e1f9a518157c0c0c637 | tserver-02.example.com | HEALTHY
e7e53a91fe704296b3a59ad304e7444a | tserver-03.example.com | UNAVAILABLE
Error from 127.0.0.1:7150: Network error: could not get status from server: Client
connection negotiation failed: client connection to 127.0.0.1:7150: connect: Connection
refused (error 61) (UNAVAILABLE)

... (full output elided)

-----
Errors:
-----
Network error: error fetching info from tablet servers: failed to gather info for all
tablet servers: 1 of 3 had errors
Corruption: table consistency check error: 1 out of 1 table(s) are not healthy

FAILED
Runtime error: ksck discovered errors

```

To verify data integrity, the optional `--checksum_scan` flag can be set, which will ensure the cluster has consistent data by scanning each tablet replica and comparing results. The `--tables` or `--tablets` flags can be used to limit the scope of the checksum scan to specific tables or tablets, respectively.

For example, checking data integrity on the `my_table` table can be done with the following command:

```

$ sudo -u kudu kudu cluster ksck --checksum_scan --tables my_table
master-01.example.com,master-02.example.com,master-03.example.com

```

By default, `ksck` will attempt to use a snapshot scan of the table, so the checksum scan can be done while writes continue.

Finally, `ksck` also supports output in JSON format using the `--ksck_format` flag. JSON output contains the same information as the plain text output, but in a format that can be used by other tools. See `kudu cluster ksck --help` for more information.

Changing Directory Configuration

For higher read parallelism and larger volumes of storage per server, you may want to configure servers to store data in multiple directories on different devices. Once a server is started, you must go through the following steps to change the directory configuration.

You can add or remove data directories to an existing master or tablet server via the `kudu fs update_dirs` tool. Data is striped across data directories, and when a new data directory is added, new data will be striped across the union of the old and new directories.



Note: Unless the `--force` flag is specified, Kudu will not allow for the removal of a directory across which tablets are configured to spread data. If `--force` is specified, all tablets configured to use that directory will fail upon starting up and be replicated elsewhere.



Note: If the [metadata directory](#) overlaps with a data directory, as was the default prior to Kudu 1.7, or if a non-default metadata directory is configured, the `--fs_metadata_dir` configuration must be specified when running the `kudu fs update_dirs` tool.



Note: Only new tablet replicas, i.e. brand new tablets' replicas and replicas that are copied to the server for high availability, will use the new directory. Existing tablet replicas on the server will not be rebalanced across the new directory.

1. The tool can only run while the server is offline, so establish a maintenance window to update the server. The tool itself runs quickly, so this offline window should be brief, and as such, only the server to update needs to be offline.

However, if the server is offline for too long (see the `follower_unavailable_considered_failed_sec` flag), the tablet replicas on it may be evicted from their Raft groups. To avoid this, it may be desirable to bring the entire cluster offline while performing the update.

2. Run the tool with the desired directory configuration flags. For example, if a cluster was set up with `--fs_wal_dir=/wals`, `--fs_metadata_dir=/meta`, and `--fs_data_dirs=/data/1, /data/2, /data/3`, and `/data/3` is to be removed (e.g. due to a disk error), run the command:

```
$ sudo -u kudu kudu fs update_dirs --force --fs_wal_dir=/wals --fs_metadata_dir=/meta --fs_data_dirs=/data/1, /data/2
```

3. Modify the values of the `fs_data_dirs` flags for the updated sever. If using Cloudera Manager, make sure to only update the configurations of the updated server, rather than of the entire Kudu service.
4. Once complete, the server process can be started. When Kudu is installed using system packages, `service` is typically used:

```
$ sudo service kudu-tserver start
```

Recovering from Disk Failure

Kudu nodes can only survive failures of disks on which certain Kudu directories are mounted. For more information about the different Kudu directory types, see the section on [Directory Configurations](#) on page 22.

The table below summarizes the resilience to disk failure in different releases of Apache Kudu.

Table 4: Kudu Disk Failure Behavior

Node Type	Kudu Directory Type	Kudu Releases that Crash on Disk Failure
Master	All	All
Tablet Server	Directory containing WALs	All
Tablet Server	Directory containing tablet metadata	All
Tablet Server	Directory containing data blocks only	Pre-1.6.0

When a disk failure occurs that does not lead to a crash, Kudu will stop using the affected directory, shut down tablets with blocks on the affected directories, and automatically re-replicate the affected tablets to other tablet servers. The affected server will remain alive and print messages to the log indicating the disk failure, for example:

```
E1205 19:06:24.163748 27115 data_dirs.cc:1011] Directory /data/8/kudu/data marked as failed
E1205 19:06:30.324795 27064 log_block_manager.cc:1822] Not using report from /data/8/kudu/data: IO error: Could not open container 0a6283cab82d4e75848f49772d2638fe: /data/8/kudu/data/0a6283cab82d4e75848f49772d2638fe.metadata: Read-only file system (error 30)
E1205 19:06:33.564638 27220 ts_tablet_manager.cc:946] T 4957808439314e0d97795c1394348d80 P 70f7ee61ead54b1885d819f354eb3405: aborting tablet bootstrap: tablet has data in a failed directory
```

While in this state, the affected node will avoid using the failed disk, leading to lower storage volume and reduced read parallelism. The administrator should schedule a brief window to [Changing Directory Configuration](#) on page 42 to exclude the failed disk.

When the disk is repaired, remounted, and ready to be reused by Kudu, take the following steps:

1. Make sure that the Kudu portion of the disk is completely empty.
2. Stop the tablet server.
3. Run the `update_dirs` tool. For example, to add `/data/3`, run the following:

```
$ sudo -u kudu kudu fs update_dirs --force --fs_wal_dir=/wals --fs_data_dirs=/data/1,/data/2,/data/3
```

4. Start the tablet server.
5. Run `ksck` to verify cluster health. For example:

```
$ sudo -u kudu kudu cluster ksck master-01.example.com
```

 **Note:** Note that existing tablets will not stripe to the restored disk, but any new tablets will stripe to the restored disk.

Recovering from Full Disks

By default, Kudu reserves a small amount of space, 1% by capacity, in its directories. Kudu considers a disk full if there is less free space available than the reservation. Kudu nodes can only tolerate running out of space on disks on which certain Kudu directories are mounted. For more information about the different Kudu directory types, see [Directory Configurations](#) on page 22. The table below describes this behavior for each type of directory. The behavior is uniform across masters and tablet servers.

Kudu Directory Type	Crash on Full Disk?
Directory containing WALs	Yes
Directory containing tablet metadata	Yes

Kudu Directory Type	Crash on Full Disk?
Directory containing data blocks only	No (see below)

Prior to Kudu 1.7.0, Kudu stripes tablet data across all directories, and will avoid writing data to full directories. Kudu will crash if all data directories are full.

In 1.7.0 and later, new tablets are assigned a disk group consisting of data directories. The number of data directories are as specified by the `-fs_target_data_dirs_per_tablet` flag with the default being 3. If Kudu is not configured with enough data directories for a full disk group, all data directories are used. When a data directory is full, Kudu will stop writing new data to it and each tablet that uses that data directory will write new data to other data directories within its group. If all data directories for a tablet are full, Kudu will crash. Periodically, Kudu will check if full data directories are still full, and will resume writing to those data directories if space has become available.

If Kudu does crash because its data directories are full, freeing space on the full directories will allow the affected daemon to restart and resume writing. Note that it may be possible for Kudu to free some space by running:

```
$ sudo -u kudu kudu fs check --repair
```

However, the above command may also fail if there is too little space left.

It's also possible to allocate additional data directories to Kudu in order to increase the overall amount of storage available. See the documentation on [updating a node's directory configuration](#) for more information. Note that existing tablets will not use new data directories, so adding a new data directory does not resolve issues with full disks.

Bringing a Tablet That Has Lost a Majority of Replicas Back Online

If a tablet has permanently lost a majority of its replicas, it cannot recover automatically and operator intervention is required. If the tablet servers hosting a majority of the replicas are down (i.e. ones reported as "TS unavailable" by `ksck`), they should be recovered instead if possible.

Suppose a tablet has lost a majority of its replicas. The first step in diagnosing and fixing the problem is to examine the tablet's state using `ksck`:

```
$ sudo -u kudu kudu cluster ksck --tablets=e822cab6c0584bc0858219d1539a17e6
master-00,master-01,master-02
Connected to the Master
Fetched info from all 5 Tablet Servers
Tablet e822cab6c0584bc0858219d1539a17e6 of table 'my_table' is unavailable: 2 replica(s)
not RUNNING
638a20403e3e4ae3b55d4d07d920e6de (tserver-00:7150): RUNNING
9a56fa85a38a4edc99c6229cba68aeaa (tserver-01:7150): bad state
  State:          FAILED
  Data state:     TABLET_DATA_READY
  Last status:    <failure message>
c311fef7708a4cf9bb11a3e4cbcaab8c (tserver-02:7150): bad state
  State:          FAILED
  Data state:     TABLET_DATA_READY
  Last status:    <failure message>
```

This output shows that, for tablet `e822cab6c0584bc0858219d1539a17e6`, the two tablet replicas on `tserver-01` and `tserver-02` failed. The remaining replica is not the leader, so the leader replica failed as well. This means the chance of data loss is higher since the remaining replica on `tserver-00` may have been lagging. In general, to accept the potential data loss and restore the tablet from the remaining replicas, divide the tablet replicas into two groups:

1. Healthy replicas: Those in `RUNNING` state as reported by `ksck`
2. Unhealthy replicas

For example, in the above `ksck` output, the replica on tablet server `tserver-00` is healthy while the replicas on `tserver-01` and `tserver-02` are unhealthy. On each tablet server with a healthy replica, alter the consensus

configuration to remove unhealthy replicas. In the typical case of 1 out of 3 surviving replicas, there will be only one healthy replica, so the consensus configuration will be rewritten to include only the healthy replica.

```
$ sudo -u kudu kudu remote_replica unsafe_change_config tserver-00:7150 <tablet-id>
<tserver-00-uuid>
```

where `<tablet-id>` is `e822cab6c0584bc0858219d1539a17e6` and `<tserver-00-uuid>` is the uuid of `tserver-00`, `638a20403e3e4ae3b55d4d07d920e6de`.

Once the healthy replicas' consensus configurations have been forced to exclude the unhealthy replicas, the healthy replicas will be able to elect a leader. The tablet will become available for writes though it will still be under-replicated. Shortly after the tablet becomes available, the leader master will notice that it is under-replicated, and will cause the tablet to re-replicate until the proper replication factor is restored. The unhealthy replicas will be tombstoned by the master, causing their remaining data to be deleted.

Rebuilding a Kudu Filesystem Layout

In the event that critical files are lost, i.e. WALs or tablet-specific metadata, all Kudu directories on the server must be deleted and rebuilt to ensure correctness. Doing so will destroy the copy of the data for each tablet replica hosted on the local server. Kudu will automatically re-replicate tablet replicas removed in this way, provided the replication factor is at least three and all other servers are online and healthy.



Note: These steps use a tablet server as an example, but the steps are the same for Kudu master servers.



Warning: If multiple nodes need their FS layouts rebuilt, wait until all replicas previously hosted on each node have finished automatically re-replicating elsewhere before continuing. Failure to do so can result in permanent data loss.

1. The first step to rebuilding a server with a new directory configuration is emptying all of the server's existing directories. For example, if a tablet server is configured with `--fs_wal_dir=/data/0/kudu-tserver-wal`, `--fs_metadata_dir=/data/0/kudu-tserver-meta`, and `--fs_data_dirs=/data/1/kudu-tserver, /data/2/kudu-tserver`, the following commands will remove the WAL directory's and data directories' contents:

```
# Note: this will delete all of the data from the local tablet server.
$ rm -rf /data/0/kudu-tserver-wal/* /data/0/kudu-tserver-meta/* /data/1/kudu-tserver/*
/data/2/kudu-tserver/*
```

2. If using Cloudera Manager, update the configurations for the rebuilt server to include only the desired directories. Make sure to only update the configurations of servers to which changes were applied, rather than of the entire Kudu service.
3. After directories are deleted, the server process can be started with the new directory configuration. The appropriate sub-directories will be created by Kudu upon starting up.

Scaling Storage on Kudu Master and Tablet Servers in the Cloud

If you find that the size of your Kudu cloud deployment has exceeded previous expectations, or you simply wish to allocate more storage to Kudu, use the following set of high-level steps as a guide to increasing storage on your Kudu master or tablet server hosts. You must work with your cluster's Hadoop administrators and the system administrators to complete this process. Replace the file paths in the following steps to those relevant to your setup.

1. Run a consistency check on the cluster hosts. For instructions, see [Monitoring Cluster Health with ksck](#) on page 40.
2. On all Kudu hosts, create a new file system with the storage capacity you require. For example, `/new/data/dir`.
3. Shutdown cluster services. For a cluster managed by Cloudera Manager cluster, see [Starting and Stopping a Cluster](#).
4. Copy the contents of your existing data directory, `/current/data/dir`, to the new filesystem at `/new/data/dir`.

5. Move your existing data directory, `/current/data/dir`, to a separate temporary location such as `/tmp/data/dir`.
6. Create a new `/current/data/dir` directory.

```
mkdir /current/data/dir
```

7. Mount `/new/data/dir` as `/current/data/dir`. Make changes to `fstab` as needed.
8. Perform steps 4-7 on all Kudu hosts.
9. Startup cluster services. For a cluster managed by Cloudera Manager cluster, see [Starting and Stopping a Cluster](#).
10. Run a consistency check on the cluster hosts. For instructions, see [Monitoring Cluster Health with ksck](#) on page 40.
11. After 10 days, if everything is in working order on all the hosts, get approval from the Hadoop administrators to remove the `/backup/data/dir` directory.

Migrating Kudu Data from One Directory to Another on the Same Host

Take the following steps to move the entire Kudu data from one directory to another.



Note:

The steps were verified on an environment where the master and the server instances were configured to write the WAL/Data to the same directory.

1. Stop the Kudu service.
2. [Modify the directory configurations](#) for the Master/Server instances.
3. Move the existing data from the old directory, to the new one.
4. Make sure the file/directory ownership is set to the `kudu` user.
5. Restart the Kudu service.
6. Run `ksck` and verify for the healthy status.

Minimizing Cluster Disruption During Temporary Planned Downtime of a Single Tablet Server

If a single tablet server is brought down temporarily in a healthy cluster, all tablets will remain available and clients will function as normal, after potential short delays due to leader elections. However, if the downtime lasts for more than `--follower_unavailable_considered_failed_sec` (default 300) seconds, the tablet replicas on the down tablet server will be replaced by new replicas on available tablet servers. This will cause stress on the cluster as tablets re-replicate and, if the downtime lasts long enough, significant reduction in the number of replicas on the down tablet server. This may require the rebalancer to fix.

To work around this, in Kudu versions 1.11 onward, the `kudu` CLI contains a tool to put tablet servers into maintenance mode. While in this state, the tablet server's replicas are not re-replicated due to its downtime alone, though re-replication may still occur in the event that the server in maintenance suffers from a disk failure or if a follower replica on the tablet server falls too far behind its leader replica. Upon exiting maintenance, re-replication is triggered for any remaining under-replicated tablets.

The `kudu tserver state enter_maintenance` and `kudu tserver state exit_maintenance` tools are added to orchestrate tablet server maintenance. The following can be run from a tablet server to put it into maintenance:

```
$ TS_UUID=$(sudo -u kudu kudu fs dump uuid --fs_wal_dir=<wal_dir>
--fs_data_dirs=<data_dirs>)
$ sudo -u kudu kudu tserver state enter_maintenance <master_addresses> "$TS_UUID"
```

The tablet server maintenance mode is shown in the "Tablet Servers" page of the Kudu leader master's web UI, and in the output of `kudu cluster ksck`. To exit maintenance mode, run the following command:

```
sudo -u kudu kudu tserver state exit_maintenance <master_addresses> "$STS_UUID"
```

In versions prior to 1.11, a different approach must be used to prevent unwanted re-replication. Increase `--follower_unavailable_considered_failed_sec` on all tablet servers so the amount of time before re-replication starts is longer than the expected downtime of the tablet server, including the time it takes the tablet server to restart and bootstrap its tablet replicas. To do this, run the following command for each tablet server:

```
$ sudo -u kudu kudu tserver set_flag <tserver_address>
follower_unavailable_considered_failed_sec <num_seconds>
```

where `<num_seconds>` is the number of seconds that will encompass the downtime. Once the downtime is finished, reset the flag to its original value.

```
$ sudo -u kudu kudu tserver set_flag <tserver_address>
follower_unavailable_considered_failed_sec <original_value>
```

On Kudu versions prior to 1.8, the `--force` flag must be provided in the above `set_flag` commands.

Running Tablet Rebalancing Tool

The `kudu` CLI contains a rebalancing tool that can be used to rebalance tablet replicas among tablet servers. For each table, the tool attempts to balance the number of replicas per tablet server. It will also, without unbalancing any table, attempt to even out the number of replicas per tablet server across the cluster as a whole.

The rebalancing tool should be run as the Kudu admin user, specifying all master addresses:

```
sudo -u kudu kudu cluster rebalance
master-01.example.com,master-02.example.com,master-03.example.com
```

When run, the rebalancer will report on the initial tablet replica distribution in the cluster, log the replicas it moves, and print a final summary of the distribution when it terminates:

```
Per-server replica distribution summary:
  Statistic | Value
-----+-----
  Minimum Replica Count | 0
  Maximum Replica Count | 24
  Average Replica Count | 14.400000

Per-table replica distribution summary:
  Replica Skew | Value
-----+-----
  Minimum | 8
  Maximum | 8
  Average | 8.000000

I0613 14:18:49.905897 3002065792 rebalancer.cc:779] tablet
e7ee9ade95b342a7a94649b7862b345d: 206a51de1486402bbb214b5ce97a633c ->
3b4d9266ac8c45ff9a5d4d7c3e1cb326 move scheduled
I0613 14:18:49.917578 3002065792 rebalancer.cc:779] tablet
5f03944529f44626a0d6ec8b1edc566e: 6e64c4165b864cbab0e67ccd82091d60 ->
ba8c22ab030346b4baa289d6d11d0809 move scheduled
I0613 14:18:49.928683 3002065792 rebalancer.cc:779] tablet
9373fee3bfe74cec9054737371a3b15d: fab382adf72c480984c6cc868fdd5f0e ->
3b4d9266ac8c45ff9a5d4d7c3e1cb326 move scheduled

... (full output elided)

I0613 14:19:01.162802 3002065792 rebalancer.cc:842] tablet
f4c046f18b174cc2974c65ac0bf52767: 206a51de1486402bbb214b5ce97a633c ->
3b4d9266ac8c45ff9a5d4d7c3e1cb326 move completed: OK
```

```

rebalancing is complete: cluster is balanced (moved 28 replicas)
Per-server replica distribution summary:
  Statistic | Value
-----+-----
  Minimum Replica Count | 14
  Maximum Replica Count | 15
  Average Replica Count | 14.400000

Per-table replica distribution summary:
  Replica Skew | Value
-----+-----
  Minimum | 1
  Maximum | 1
  Average | 1.000000
    
```

If more details are needed in addition to the replica distribution summary, use the `--output_replica_distribution_details` flag. If added, the flag makes the tool print per-table and per-tablet server replica distribution statistics as well.

Use the `--report_only` flag to get a report on table-wide and cluster-wide replica distribution statistics without starting any rebalancing activity.

The rebalancer can also be restricted to run on a subset of the tables by supplying the `--tables` flag. Note that, when running on a subset of tables, the tool will not attempt to balance the cluster as a whole.

The length of time rebalancing is run for can be controlled with the flag `--max_run_time_sec`. By default, the rebalancer will run until the cluster is balanced. To control the amount of resources devoted to rebalancing, modify the flag `--max_moves_per_server`. See `kudu cluster rebalance --help` for more.

It's safe to stop the rebalancer tool at any time. When restarted, the rebalancer will continue rebalancing the cluster.

The rebalancer tool requires all registered tablet servers to be up and running to proceed with the rebalancing process in order to avoid possible conflicts and races with the automatic re-replication and to keep replica placement optimal for current configuration of the cluster. If a tablet server becomes unavailable during the rebalancing session, the rebalancer will exit. As noted above, it's safe to restart the rebalancer after resolving the issue with unavailable tablet servers.

The rebalancing tool can rebalance Kudu clusters running older versions as well, with some restrictions. Consult the following table for more information. In the table, "RF" stands for "replication factor".

Version Range	Rebalances RF = 1 Tables?	Rebalances RF > 1 Tables?
v < 1.4.0	No	No
1.4.0 <= v < 1.7.1	No	Yes
v >= 1.7.1	Yes	Yes

If the rebalancer is running against a cluster where rebalancing replication factor one tables is not supported, it will rebalance all the other tables and the cluster as if those singly-replicated tables did not exist.

Running Tablet Rebalancing Tool on Rack-Aware Cluster

As detailed in the [Rack Awareness \(Location Awareness\)](#) on page 28 section, it's possible to use the `kudu cluster rebalance` tool to establish the placement policy on a cluster. This might be necessary when the rack awareness feature is first configured or when re-replication violated the placement policy. The rebalancing tool breaks its work into three phases:

1. The rack-aware rebalancer tries to establish the placement policy. Use the `--disable_policy_fixer` flag to skip this phase.
2. The rebalancer tries to balance load by location, moving tablet replicas between locations in an attempt to spread tablet replicas among locations evenly. The load of a location is measured as the total number of replicas in the

location divided by the number of tablet servers in the location. Use the `--disable_cross_location_rebalancing` flag to skip this phase.

3. The rebalancer tries to balance the tablet replica distribution within each location, as if the location were a cluster on its own. Use the `--disable_intra_location_rebalancing` flag to skip this phase.

By using the `--report_only` flag, it's also possible to check if all tablets in the cluster conform to the placement policy without attempting any replica movement.

Running Tablet Rebalancing Tool in Cloudera Manager

To run the tablet rebalancer tool in Cloudera Manager:

1. Browse to **Clusters > Kudu**.
2. Click **Actions** and select **Run Kudu Rebalancer Tool**.

In Cloudera Manager, the rebalancer runs with the default flags.

Decommissioning or Permanently Removing a Tablet Server From a Cluster

Kudu does not currently support an automated way to remove a tablet server from a cluster permanently. Use the following steps to manually remove a tablet server:

1. Ensure the cluster is in good health using `ksck`. See [Checking Cluster Health with `ksck`](#).
2. If the tablet server contains any replicas of tables with replication factor 1, these replicas must be manually moved off the tablet server prior to shutting it down. Use the `kudu tablet change_config move_replica` tool for this.
3. Shut down the tablet server. After `-follower_unavailable_considered_failed_sec`, which defaults to 5 minutes, Kudu will begin to re-replicate the tablet server's replicas to other servers. Wait until the process is finished. Progress can be monitored using `ksck`.
4. Once all the copies are complete, `ksck` will continue to report the tablet server as unavailable. The cluster will otherwise operate fine without the tablet server. To completely remove it from the cluster so `ksck` shows the cluster as completely healthy, restart the masters. In the case of a single master, this will cause cluster downtime. With multi-master, restart the masters in sequence to avoid cluster downtime.



Important: Do not shut down multiple tablet servers at once. To remove multiple tablet servers from the cluster, follow the above instructions for each tablet server, ensuring that the previous tablet server is removed from the cluster and `ksck` is healthy before shutting down the next.

Using cluster names in the `kudu` command line tool

When using the `kudu` command line tool, it can be difficult to remember the precise list of Kudu master RPC addresses needed to communicate with a cluster, especially when managing multiple clusters. As an alternative, the command line tool can identify clusters by name. To use this functionality:

1. Create a new directory to store the Kudu configuration file.
2. Export the path to this directory in the `KUDU_CONFIG` environment variable.
3. Create a file called `kudurc` in the new directory.
4. Populate `kudurc` as follows, substituting your own cluster names and RPC addresses:

```
clusters_info:
  cluster_name1:
    master_addresses: ip1:port1,ip2:port2,ip3:port3
  cluster_name2:
    master_addresses: ip4:port4
```

5. When using the `kudu` command line tool, replace the list of Kudu master RPC addresses with the cluster name, prepended with the character `@`.

```
$ sudo -u kudu kudu ksck @cluster_name1
```



Note: Cluster names may be used as input in any invocation of the `kudu` command line tool that expects a list of Kudu master RPC addresses.

Managing Kudu Using Cloudera Manager

This topic describes the tasks you can perform to manage the Kudu service using Cloudera Manager. You can use Cloudera Manager to upgrade the Kudu service, start and stop the Kudu service, monitor operations, and configure the Kudu master and tablet servers, among other tasks. Depending on your deployment, there are several different configuration settings you may need to modify.

For detailed information about Apache Kudu, view the [Apache Kudu Guide](#).

Installing and Upgrading the Kudu Service

You can install Kudu through the Cloudera Manager installation wizard, using either parcels or packages. For instructions, see [Cloudera Installation Guide](#).

Enabling Core Dump for the Kudu Service

If Kudu crashes, you can use Cloudera Manager to generate a core dump to get more information about the crash.

1. Go to the Kudu service.
2. Click the **Configuration** tab.
3. Search for `core dump`.
4. Check the checkbox for the **Enable Core Dump** property.
5. (Optional) Unless otherwise configured, the dump file is generated in the default core dump directory, `/var/log/kudu`, for both the Kudu master and the tablet servers.
 - To configure a different dump directory for the Kudu master, modify the value of the **Kudu Master Core Dump Directory** property.
 - To configure a different dump directory for the Kudu tablet servers, modify the value of the **Kudu Tablet Server Core Dump Directory** property.
6. Click **Save Changes**.

Verifying the Impala Dependency on Kudu

In a Cloudera Manager deployment, once the Kudu service is installed, Impala will automatically identify the Kudu Master. However, if your Impala queries don't work as expected, use the following steps to make sure that the Impala service is set to be dependent on Kudu.

1. Go to the Impala service.
2. Click the **Configuration** tab.
3. Search for `kudu`.
4. Make sure the **Kudu Service** property is set to the right Kudu service.
5. Click **Save Changes**.

Using the Charts Library with the Kudu Service

By default, the **Status** tab for the Kudu service displays a dashboard containing a limited set of charts. For details on the terminology used in these charts, and instructions on how to query for time-series data, display chart details, and edit charts, see [Charting Time-Series Data](#).

The Kudu service's Charts Library tab also displays a dashboard containing a much larger set of charts, organized by categories such as process charts, host charts, CPU charts, and so on, depending on the entity (service, role, or host)

that you are viewing. You can use these charts to keep track of disk space usage, the rate at which data is being inserted/modified in Kudu across all tables, or any critical cluster events. You can also use them to keep track of individual tables. For example, to find out how much space a Kudu table is using on disk:

1. Go to the Kudu service and navigate to the **Charts Library** tab.
2. On the left-hand side menu, click **Tables** to display the list of tables currently stored in Kudu.
3. Click on a table name to view the default dashboard for that table. The **Total Tablet Size On Disk Across Kudu Replicas** chart displays the total size of the table on disk using a time-series chart.

Hovering with your mouse over the line on the chart opens a small pop-up window that displays information about that data point. Click the data stream within the chart to display a larger pop-up window that includes additional information for the table at the point in time where the mouse was clicked.

Developing Applications With Apache Kudu

Apache Kudu provides C++ and Java client APIs, as well as reference examples to illustrate their use.



Warning: Use of server-side or private interfaces is not supported, and interfaces which are not part of public APIs have no stability guarantees.

Viewing the API Documentation

C++ API Documentation

The documentation for the C++ client APIs is included in the header files in `/usr/include/kudu/` if you installed Kudu using packages or subdirectories of `src/kudu/client/` if you built Kudu from source. If you installed Kudu using parcels, no headers are included in your installation, and you will need to build Kudu from source in order to have access to the headers and shared libraries.

The following command is a naive approach to finding relevant header files. Use of any APIs other than the client APIs is unsupported.

```
find /usr/include/kudu -type f -name *.h
```

Java API Documentation

View the [Java API documentation](#) online. Alternatively, after building the Java client, Java API documentation is available in `java/kudu-client/target/apidocs/index.html`.

Kudu Example Applications

Several example applications are provided in the [examples directory](#) of the Apache Kudu git repository. Each example includes a `README` that shows how to compile and run it. The following list includes some of the examples that are available today. Check the repository itself in case this list goes out of date.

cpp/example.cc

A simple C++ application which connects to a Kudu instance, creates a table, writes data to it, then drops the table.

java/java-example

A simple Java application which connects to a Kudu instance, creates a table, writes data to it, then drops the table.

java/insert-loadgen

A small Java application which listens on a TCP socket for time series data corresponding to the Collectl wire protocol. The commonly-available collectl tool can be used to send example data to the server.

python/dstat-kudu

A Java application that generates random insert load.

python/graphite-kudu

An example program that shows how to use the Kudu Python API to load data into a new / existing Kudu table generated by an external program, `dstat` in this case.

python/graphite-kudu

An example plugin for using graphite-web with Kudu as a backend.

These examples should serve as helpful starting points for your own Kudu applications and integrations.

Maven Artifacts

The following Maven `<dependency>` element is valid for the Apache Kudu GA release:

```
<dependency>
  <groupId>org.apache.kudu</groupId>
  <artifactId>kudu-client</artifactId>
  <version>1.9.0-cdh6.2.0</version>
</dependency>
```

Convenience binary artifacts for the Java client and various Java integrations (e.g. Spark, Flume) are also now available via the [ASF Maven repository](#) and the [Central Maven repository](#).

Building the Java Client

Requirements

- JDK 7
- Apache Maven 3.x
- `protoc` 2.6 or newer installed in your path, or built from the `thirdparty/` directory. Run the following commands to build `protoc` from the third-party dependencies:

```
thirdparty/download-thirdparty.sh
thirdparty/build-thirdparty.sh protocbuf
```

To build the Java client, clone the Kudu Git repository, change to the `java` directory, and issue the following command:

```
mvn install -DskipTests
```

For more information about building the Java API, as well as Eclipse integration, see `java/README.md`.

Kudu Python Client

The Kudu Python client provides a Python friendly interface to the C++ client API.

To install and use the Kudu Python client, you need to install the Kudu C++ client libraries and headers. See [Install Using Packages](#) topic for installing Kudu C++ client.



Note:

Starting CDH 6.3.3, you must have an Enterprise Support Subscription to access the archive.cloudera.com/p/ repository.

To install the Kudu Python client:

1. Update all the packages on your system using the following command: `sudo yum -y update`
2. Install the extra packages for the Enterprise Linux distribution: `sudo yum -y install epel-release`
3. Install the Python package manager: `sudo yum -y install python-pip`
4. Verify the version of the PIP manager that you just installed: `pip --version`
5. Install Cython: `sudo pip install cython`
6. Download the following files:

- wget
http://username:password@archive.cloudera.com/p/cdh6/6.3.3/redhat7/yum/RPM/x86_64/kudu-1.10.0+cdh6.3.x-1822539.x86_64.rpm
- wget
http://username:password@archive.cloudera.com/p/cdh6/6.3.3/redhat7/yum/RPM/x86_64/kudu-client-devel-1.10.0+cdh6.3.x-1822539.x86_64.rpm

7. Install the kudu package from the local directory: `sudo yum -y localinstall ./kudu-*`
8. Install the package used for developing Python extensions: `sudo yum -y install python-devel`
9. Upgrade the setup tools: `sudo pip install --upgrade pip setuptools`
10. Install the C++ compiler: `sudo yum -y install gcc-c++`
11. Install the Kudu-python client: `sudo pip install kudu-python==1.10.0`
12. Install kudu-python: `sudo pip install kudu-python`

The sample below demonstrates the use of part of the Python client.

```
import kudu
from kudu.client import Partitioning
from datetime import datetime

# Connect to Kudu master server
client = kudu.connect(host='kudu.master', port=7051)

# Define a schema for a new table
builder = kudu.schema_builder()
builder.add_column('key').type(kudu.int64).nullable(False).primary_key()
builder.add_column('ts_val', type_=kudu.unixtime_micros, nullable=False,
compression='lz4')
schema = builder.build()

# Define partitioning schema
partitioning = Partitioning().add_hash_partitions(column_names=['key'], num_buckets=3)

# Create new table
client.create_table('python-example', schema, partitioning)

# Open a table
table = client.table('python-example')

# Create a new session so that we can apply write operations
session = client.new_session()

# Insert a row
op = table.new_insert({'key': 1, 'ts_val': datetime.utcnow()})
session.apply(op)

# Upsert a row
op = table.new_upsert({'key': 2, 'ts_val': "2016-01-01T00:00:00.000000"})
session.apply(op)

# Updating a row
op = table.new_update({'key': 1, 'ts_val': ("2017-01-01", "%Y-%m-%d")})
session.apply(op)

# Delete a row
op = table.new_delete({'key': 2})
session.apply(op)

# Flush write operations, if failures occur, capture print them.
try:
    session.flush()
except kudu.KuduBadStatus as e:
    print(session.get_pending_errors())

# Create a scanner and add a predicate
scanner = table.scanner()
scanner.add_predicate(table['ts_val'] == datetime(2017, 1, 1))

# Open Scanner and read all tuples
# Note: This doesn't scale for large scans
result = scanner.open().read_all_tuples()
```

Example Apache Impala Commands With Kudu

See [Using Apache Impala with Kudu](#) on page 60 for guidance on installing and using Impala with Kudu, including several `impala-shell` examples.

Kudu Integration with Spark

Kudu integrates with Spark through the Data Source API as of version 1.0.0. Include the `kudu-spark` dependency using the `--packages` option.

Use the `kudu-spark_2.10` artifact if using Spark with Scala 2.10. Note that Spark 1 is no longer supported in Kudu starting from version 1.6.0. So in order to use Spark 1 integrated with Kudu, version 1.5.0 is the latest to go to.

```
spark-shell --packages org.apache.kudu:kudu-spark_2.10:1.5.0-cdh5.13.1 --repositories
https://repository.cloudera.com/artifactory/cloudera-repos/
```

Use `kudu-spark2_2.11` artifact if using Spark 2 with Scala 2.11.



Note: `kudu-spark` versions 1.8.0 and below have slightly different syntax. See the documentation of your version for a valid example. Versioned documentation can be found on the [releases page](#).

```
spark2-shell --packages org.apache.kudu:kudu-spark2_2.11:1.9.0-cdh6.2.0 --repositories
https://repository.cloudera.com/artifactory/cloudera-repos/
```

Below is a minimal Spark SQL `select` example. You first import the `kudu spark` package, then create a `DataFrame`, and then create a view from the `DataFrame`. After those steps, the table is accessible from Spark SQL. You can also refer to the Spark [quickstart guide](#) or this [Kudu-Spark example](#).



Note: You can use the Kudu CLI tool to create table and generate data by `kudu perf loadgen kudu.master:7051 -keep_auto_table` for the following two examples:

```
import org.apache.kudu.spark.kudu._

// Create a DataFrame that points to the Kudu table we want to query.
val df = spark.read.options(Map("kudu.master" -> "kudu.master:7051",
                               "kudu.table" -> "default.my_table")).format("kudu").load
// Create a view from the DataFrame to make it accessible from Spark SQL.
df.createOrReplaceTempView("my_table")
// Now we can run Spark SQL queries against our view of the Kudu table.
spark.sql("select * from my_table").show()
```

Below is a more advanced example that includes both reads and writes:

```
import org.apache.kudu.client._
import org.apache.kudu.spark.kudu.KuduContext
import collection.JavaConverters._

// Read a table from Kudu
val df = spark.read
  .options(Map("kudu.master" -> "kudu.master:7051", "kudu.table" -> "kudu_table"))
  .format("kudu").load

// Query using the Spark API...
df.select("key").filter("key >= 5").show()

// ...or register a temporary table and use SQL
df.createOrReplaceTempView("kudu_table")
val filteredDF = spark.sql("select key from kudu_table where key >= 5").show()
```

```

// Use KuduContext to create, delete, or write to Kudu tables
val kuduContext = new KuduContext("kudu.master:7051", spark.sparkContext)

// Create a new Kudu table from a DataFrame schema
// NB: No rows from the DataFrame are inserted into the table
kuduContext.createTable(
  "test_table", df.schema, Seq("key"),
  new CreateTableOptions()
    .setNumReplicas(1)
    .addHashPartitions(List("key").asJava, 3))

// Check for the existence of a Kudu table
kuduContext.tableExists("test_table")

// Insert data
kuduContext.insertRows(df, "test_table")

// Delete data
kuduContext.deleteRows(df, "test_table")

// Upsert data
kuduContext.upsertRows(df, "test_table")

// Update data
val updateDF = df.select($"key", ($"int_val" + 1).as("int_val"))
kuduContext.updateRows(updateDF, "test_table")

// Data can also be inserted into the Kudu table using the data source, though the
// methods on
// KuduContext are preferred
// NB: The default is to upsert rows; to perform standard inserts instead, set operation
// = insert
// in the options map
// NB: Only mode Append is supported

df.write
  .options(Map("kudu.master" -> "kudu.master:7051", "kudu.table" -> "test_table"))
  .mode("append")
  .format("kudu").save

// Delete a Kudu table
kuduContext.deleteTable("test_table")

```

Upsert Option in Kudu Spark

The upsert operation in kudu-spark supports an extra write option of `ignoreNull`. If set to true, it will avoid setting existing column values in Kudu table to Null if the corresponding DataFrame column values are Null. If unspecified, `ignoreNull` is false by default.

```

val dataDataFrame = spark.read
  .options(Map("kudu.master" -> "kudu.master:7051", "kudu.table" -> simpleTableName))
  .format("kudu").load

dataDataFrame.registerTempTable(simpleTableName)
dataDataFrame.show()

// Below is the original data in the table 'simpleTableName'
+----+----+
|key|val|
+----+----+
|  0|foo|
+----+----+

// Upsert a row with existing key 0 and val Null with ignoreNull set to true
val nullDF = spark.createDataFrame(Seq((0, null.asInstanceOf[String]))).toDF("key",
"val")
val wo = new KuduWriteOptions
wo.ignoreNull = true
kuduContext.upsertRows(nullDF, simpleTableName, wo)
dataDataFrame.show()
// The val field stays unchanged

```

```
+---+---+
|key|val|
+---+---+
| 0|foo|
+---+---+

// Upsert a row with existing key 0 and val Null with ignoreNull default/set to false
kuduContext.upsertRows(nullDF, simpleTableName)
// Equivalent to:
// val wo = new KuduWriteOptions
// wo.ignoreNull = false
// kuduContext.upsertRows(nullDF, simpleTableName, wo)
df.show()
// The val field is set to Null this time
+---+---+
|key| val|
+---+---+
| 0|null|
+---+---+
```

Using Spark with a Secure Kudu Cluster

The Kudu-Spark integration is able to operate on secure Kudu clusters which have authentication and encryption enabled, but the submitter of the Spark job must provide the proper credentials. For Spark jobs using the default 'client' deploy mode, the submitting user must have an active Kerberos ticket granted through `kinit`. For Spark jobs using the 'cluster' deploy mode, a Kerberos principal name and keytab location must be provided through the `--principal` and `--keytab` arguments to `spark2-submit`.

Spark Integration Known Issues and Limitations

- Spark 2.2 (and higher) requires Java 8 at runtime even though Kudu Spark 2.x integration is Java 7 compatible. Spark 2.2 is the default dependency version as of Kudu 1.5.0.
- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when registered as a temporary table.
- Kudu tables with a column name containing upper case or non-ASCII characters must not be used with SparkSQL. Columns can be renamed in Kudu to work around this issue.
- `<>` and `OR` predicates are not pushed to Kudu, and instead will be evaluated by the Spark task. Only `LIKE` predicates with a suffix wildcard are pushed to Kudu. This means `LIKE "FOO%"` will be pushed, but `LIKE "FOO%BAR"` won't.
- Kudu does not support all the types supported by Spark SQL. For example, `Date` and complex types are not supported.
- Kudu tables can only be registered as temporary tables in SparkSQL.
- Kudu tables cannot be queried using `HiveContext`.

Spark Integration Best Practices

Avoid multiple Kudu clients per cluster

One common Kudu-Spark coding error is instantiating extra `KuduClient` objects. In `kudu-spark`, a `KuduClient` is owned by the `KuduContext`. Spark application code should not create another `KuduClient` connecting to the same cluster. Instead, application code should use the `KuduContext` to access a `KuduClient` using `KuduContext#syncClient`.

To diagnose multiple `KuduClient` instances in a Spark job, look for signs in the logs of the master being overloaded by many `GetTableLocations` or `GetTabletLocations` requests coming from different clients, usually around the same time. This symptom is especially likely in Spark Streaming code, where creating a `KuduClient` per task will result in periodic waves of master requests from new clients.

Integration with MapReduce, YARN, and Other Frameworks

Kudu was designed to integrate with MapReduce, YARN, Spark, and other frameworks in the Hadoop ecosystem. See [RowCounter.java](#) and [ImportCsv.java](#) for examples which you can model your own integrations on.

Using Apache Impala with Kudu

Apache Kudu has tight integration with Apache Impala, allowing you to use Impala to insert, query, update, and delete data from Kudu tablets using Impala's SQL syntax, as an alternative to using the Kudu APIs to build a custom Kudu application. In addition, you can use JDBC or ODBC to connect existing or new applications written in any language, framework, or business intelligence tool to your Kudu data, using Impala as the broker.

Prerequisites

- To use Impala to query Kudu data as described in this topic, you will require Cloudera Manager and CDH 5.10.x or higher.
- The syntax described in this topic is specific to Impala included in CDH 5.10 and higher, and will not work on previous versions. If you are using a lower version of Impala (including the `IMPALA_KUDU` releases previously available), upgrade to CDH 5.10 or higher.

Note that this topic does not describe Impala installation or upgrade procedures. Refer to the [Impala](#) documentation to make sure you are able to run queries against Impala tables on HDFS before proceeding.

- Lower versions of CDH and Cloudera Manager used an experimental fork of Impala which is referred to as `IMPALA_KUDU`. If you have previously installed the `IMPALA_KUDU` service, make sure you remove it from your cluster before you proceed. Install Kudu 1.2.x (or higher) using [Cloudera Manager](#).

Impala Database Containment Model

Every Impala table is contained within a namespace called a database. The default database is called `default`, and you may create and drop additional databases as desired. To create the database, use a `CREATE DATABASE` statement. To use the database for further Impala operations such as `CREATE TABLE`, use the `USE` statement. For example, to create a table in a database called `impala_kudu`, use the following statements:

```
CREATE DATABASE impala_kudu;
USE impala_kudu;
CREATE TABLE my_first_table (
...

```

The `my_first_table` table is created within the `impala_kudu` database.

The prefix `impala::` and the Impala database name are appended to the underlying Kudu table name:
`impala::<database>.<table>`

For example, to specify the `my_first_table` table in database `impala_kudu`, as opposed to any other table with the same name in another database, refer to the table as `impala::impala_kudu.my_first_table`. This also applies to `INSERT`, `UPDATE`, `DELETE`, and `DROP` statements.

Internal and External Impala Tables

When creating a new Kudu table using Impala, you can create the table as an internal table or an external table.

Internal

An internal table (created by `CREATE TABLE`) is managed by Impala, and can be dropped by Impala. When you create a new table using Impala, it is generally a internal table. When such a table is created in Impala, the corresponding Kudu table will be named `impala::database_name.table_name`. The prefix is always `impala::`, and the database name and table name follow, separated by a dot.

External

An external table (created by `CREATE EXTERNAL TABLE`) is not managed by Impala, and dropping such a table does not drop the table from its source location (here, Kudu). Instead, it only removes the mapping between Impala and Kudu. This is the mode used in the syntax provided by Kudu for mapping an existing table to Impala.

See the [Impala documentation](#) for more information about internal and external tables.

Using Impala To Query Kudu Tables

Neither Kudu nor Impala need special configuration in order for you to use the Impala Shell or the Impala API to insert, update, delete, or query Kudu data using Impala. However, you do need to create a mapping between the Impala and Kudu tables. Kudu provides the Impala query to map to an existing Kudu table in the web UI.

- Make sure you are using the `impala-shell` binary provided by the default CDH Impala binary. The following example shows how you can verify this using the `alternatives` command on a RHEL 6 host. Do not copy and paste the `alternatives --set` command directly, because the file names are likely to differ.

```
$ sudo alternatives --display impala-shell

impala-shell - status is auto.
link currently points to
/opt/cloudera/parcels/CDH-5.10.0-1.cdh5.10.0.p0.25/bin/impala-shell
/opt/cloudera/parcels/CDH-5.10.0-1.cdh5.10.0.p0.25/bin/impala-shell - priority 10
Current `best' version is
/opt/cloudera/parcels/CDH-5.10.0-1.cdh5.10.0.p0.25/bin/impala-shell.
```

- Although not necessary, it is recommended that you configure Impala with the locations of the Kudu Masters using the `--kudu_master_hosts=<master1>[:port]` flag. If this flag is not set, you will need to manually provide this configuration each time you create a table by specifying the `kudu.master_addresses` property inside a `TBLPROPERTIES` clause. If you are using Cloudera Manager, no such configuration is needed. The Impala service will automatically [recognize the Kudu Master hosts](#).

The rest of this guide assumes that this configuration has been set.

- Start Impala Shell using the `impala-shell` command. By default, `impala-shell` attempts to connect to the Impala daemon on `localhost` on port 21000. To connect to a different host, use the `-i <host:port>` option. To automatically connect to a specific Impala database, use the `-d <database>` option. For instance, if all your Kudu tables are in Impala in the database `impala_kudu`, use `-d impala_kudu` to use this database.
- To quit the Impala Shell, use the following command: `quit;`

Querying an Existing Kudu Table from Impala

Tables created through the Kudu API or other integrations such as Apache Spark are not automatically visible in Impala. To query them, you must first create an external table within Impala to map the Kudu table into an Impala database:

```
CREATE EXTERNAL TABLE my_mapping_table
STORED AS KUDU
TBLPROPERTIES (
  'kudu.table_name' = 'my_kudu_table'
);
```

Creating a New Kudu Table From Impala

Creating a new table in Kudu from Impala is similar to mapping an existing Kudu table to an Impala table, except that you need to specify the schema and partitioning information yourself. Use the examples in this section as a guideline. Impala first creates the table, then creates the mapping.

In the `CREATE TABLE` statement, the columns that comprise the primary key must be listed first. Additionally, primary key columns are implicitly considered `NOT NULL`.

When creating a new table in Kudu, you must define a partition schema to pre-split your table. The best partition schema to use depends upon the structure of your data and your data access patterns. The goal is to maximize parallelism and use all your tablet servers evenly. For more information on partition schemas, see [Partitioning Tables](#) on page 62.



Note: In Impala included in CDH 5.13 and higher, the `PARTITION BY` clause is optional for Kudu tables. If the clause is omitted, Impala automatically constructs a single partition that is not connected to any column. Because such a table cannot take advantage of Kudu features for parallelized queries and query optimizations, omitting the `PARTITION BY` clause is only appropriate for small lookup tables.

The following `CREATE TABLE` example distributes the table into 16 partitions by hashing the `id` column, for simplicity.

```
CREATE TABLE my_first_table
(
  id BIGINT,
  name STRING,
  PRIMARY KEY(id)
)
PARTITION BY HASH PARTITIONS 16
STORED AS KUDU;
```

By default, Kudu tables created through Impala use a tablet replication factor of 3. To specify the replication factor for a Kudu table, add a `TBLPROPERTIES` clause to the `CREATE TABLE` statement as shown below where `n` is the replication factor you want to use:

```
TBLPROPERTIES ('kudu.num_tablet_replicas' = 'n')
```

A replication factor must be an odd number.

Changing the `kudu.num_tablet_replicas` table property using the `ALTER TABLE` currently has no effect.

The Impala SQL Reference [CREATE TABLE](#) topic has more details and examples.

CREATE TABLE AS SELECT

You can create a table by querying any other table or tables in Impala, using a `CREATE TABLE ... AS SELECT` statement. The following example imports all rows from an existing table, `old_table`, into a new Kudu table, `new_table`. The columns in `new_table` will have the same names and types as the columns in `old_table`, but you will need to additionally specify the primary key and partitioning schema.

```
CREATE TABLE new_table
PRIMARY KEY (ts, name)
PARTITION BY HASH(name) PARTITIONS 8
STORED AS KUDU
AS SELECT ts, name, value FROM old_table;
```

You can refine the `SELECT` statement to only match the rows and columns you want to be inserted into the new table. You can also rename the columns by using syntax like `SELECT name as new_col_name`.

Partitioning Tables

Tables are partitioned into tablets according to a partition schema on the primary key columns. Each tablet is served by at least one tablet server. Ideally, a table should be split into tablets that are distributed across a number of tablet servers to maximize parallel operations. The details of the partitioning schema you use will depend entirely on the type of data you store and how you access it.

Kudu currently has no mechanism for splitting or merging tablets after the table has been created. Until this feature has been implemented, you must provide a partition schema for your table when you create it. When designing your tables, consider using primary keys that will allow you to partition your table into tablets which grow at similar rates.

You can partition your table using Impala's `PARTITION BY` clause, which supports distribution by `RANGE` or `HASH`. The partition scheme can contain zero or more `HASH` definitions, followed by an optional `RANGE` definition. The `RANGE`

definition can refer to one or more primary key columns. Examples of basic and advanced partitioning are shown below.



Note: In Impala included in CDH 5.13 and higher, the `PARTITION BY` clause is optional for Kudu tables. If the clause is omitted, Impala automatically constructs a single partition that is not connected to any column. Because such a table cannot take advantage of Kudu features for parallelized queries and query optimizations, omitting the `PARTITION BY` clause is only appropriate for small lookup tables.

Monotonically Increasing Values - If you partition by range on a column whose values are monotonically increasing, the last tablet will grow much larger than the others. Additionally, all data being inserted will be written to a single tablet at a time, limiting the scalability of data ingest. In that case, consider distributing by `HASH` instead of, or in addition to, `RANGE`.



Note: Impala keywords, such as `group`, are enclosed by back-tick characters when they are used as identifiers, rather than as keywords.

Basic Partitioning

PARTITION BY RANGE

You can specify range partitions for one or more primary key columns. Range partitioning in Kudu allows splitting a table based on specific values or ranges of values of the chosen partition keys. This allows you to balance parallelism in writes with scan efficiency.

For instance, if you have a table that has the columns `state`, `name`, and `purchase_count`, and you partition the table by `state`, it will create 50 tablets, one for each US state.

```
CREATE TABLE customers (
  state STRING,
  name STRING,
  purchase_count int,
  PRIMARY KEY (state, name)
)
PARTITION BY RANGE (state)
(
  PARTITION VALUE = 'al',
  PARTITION VALUE = 'ak',
  PARTITION VALUE = 'ar',
  ...
  PARTITION VALUE = 'wv',
  PARTITION VALUE = 'wy'
)
STORED AS KUDU;
```

PARTITION BY HASH

Instead of distributing by an explicit range, or in combination with range distribution, you can distribute into a specific number of partitions by hash. You specify the primary key columns you want to partition by, and the number of partitions you want to use. Rows are distributed by hashing the specified key columns. Assuming that the values being hashed do not themselves exhibit significant skew, this will serve to distribute the data evenly across all partitions.

You can specify multiple definitions, and you can specify definitions which use compound primary keys. However, one column cannot be mentioned in multiple hash definitions. Consider two columns, `a` and `b`:

- `HASH(a), HASH(b)` -- will succeed
- `HASH(a,b)` -- will succeed
- `HASH(a), HASH(a,b)` -- will fail



Note: `PARTITION BY HASH` with no column specified is a shortcut to create the desired number of partitions by hashing all primary key columns.

Hash partitioning is a reasonable approach if primary key values are evenly distributed in their domain and no data skew is apparent, such as timestamps or serial IDs.

The following example creates 16 tablets by hashing the `id` column. A maximum of 16 tablets can be written to in parallel. In this example, a query for a range of `sku` values is likely to need to read from all 16 tablets, so this may not be the optimum schema for this table. See [Advanced Partitioning](#) on page 64 for an extended example.

```
CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT,
  PRIMARY KEY (id, sku)
)
PARTITION BY HASH PARTITIONS 16
STORED AS KUDU;
```

Advanced Partitioning

You can combine `HASH` and `RANGE` partitioning to create more complex partition schemas. You can also specify zero or more `HASH` definitions, followed by zero or one `RANGE` definitions. Each schema definition can encompass one or more columns. While enumerating every possible distribution schema is out of the scope of this topic, the following examples illustrate some of the possibilities.

`PARTITION BY HASH` and `RANGE`

Consider the basic `PARTITION BY HASH` example above. If you often query for a range of `sku` values, you can optimize the example by combining hash partitioning with range partitioning.

The following example still creates 16 tablets, by first hashing the `id` column into 4 partitions, and then applying range partitioning to split each partition into four tablets, based upon the value of the `sku` string. At least four tablets (and possibly up to 16) can be written to in parallel, and when you query for a contiguous range of `sku` values, there's a good chance you only need to read a quarter of the tablets to fulfill the query.

By default, the entire primary key (`id, sku`) will be hashed when you use `PARTITION BY HASH`. To hash on only part of the primary key, and use a range partition on the rest, use the syntax demonstrated below.

```
CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT,
  PRIMARY KEY (id, sku)
)
```

```

PARTITION BY HASH (id) PARTITIONS 4,
RANGE (sku)
(
  PARTITION VALUES < 'g',
  PARTITION 'g' <= VALUES < 'o',
  PARTITION 'o' <= VALUES < 'u',
  PARTITION 'u' <= VALUES
)
STORED AS KUDU;

```

Multiple PARTITION BY HASH Definitions

Once again expanding on the example above, let's assume that the pattern of incoming queries will be unpredictable, but you still want to ensure that writes are spread across a large number of tablets. You can achieve maximum distribution across the entire primary key by hashing on both primary key columns.

```

CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT,
  PRIMARY KEY (id, sku)
)
PARTITION BY HASH (id) PARTITIONS 4,
              HASH (sku) PARTITIONS 4
STORED AS KUDU;

```

The example creates 16 partitions. You could also use `HASH (id, sku) PARTITIONS 16`. However, a scan for `sku` values would almost always impact all 16 partitions, rather than possibly being limited to 4.

Non-Covering Range Partitions

Kudu supports the use of non-covering range partitions, which can be used to address the following scenarios:

- In the case of time-series data or other schemas which need to account for constantly-increasing primary keys, tablets serving old data will be relatively fixed in size, while tablets receiving new data will grow without bounds.
- In cases where you want to partition data based on its category, such as sales region or product type, without non-covering range partitions you must know all of the partitions ahead of time or manually recreate your table if partitions need to be added or removed, such as the introduction or elimination of a product type.



Note: See [Range Partitioning](#) on page 89 for the caveats of non-covering range partitions.

The following example creates a tablet per year (5 tablets total), for storing log data. The table only accepts data from 2012 to 2016. Keys outside of these ranges will be rejected.

```

CREATE TABLE sales_by_year (
  year INT, sale_id INT, amount INT,
  PRIMARY KEY (sale_id, year)
)
PARTITION BY RANGE (year) (
  PARTITION VALUE = 2012,
  PARTITION VALUE = 2013,
  PARTITION VALUE = 2014,
  PARTITION VALUE = 2015,
  PARTITION VALUE = 2016
)

```

```
)  
STORED AS KUDU;
```

When records start coming in for 2017, they will be rejected. At that point, the 2017 range should be added as follows:

```
ALTER TABLE sales_by_year ADD RANGE PARTITION VALUE = 2017;
```

In use cases where a rolling window of data retention is required, range partitions may also be dropped. For example, if data from 2012 should no longer be retained, it may be deleted in bulk:

```
ALTER TABLE sales_by_year DROP RANGE PARTITION VALUE = 2012;
```

Note that just like dropping a table, this irrecoverably deletes all data stored in the dropped partition.

Partitioning Guidelines

- For large tables, such as fact tables, aim for as many tablets as you have cores in the cluster.
- For small tables, such as dimension tables, ensure that each tablet is at least 1 GB in size.

In general, be mindful the number of tablets limits the parallelism of reads, in the current implementation. Increasing the number of tablets significantly beyond the number of cores is likely to have diminishing returns.

Optimizing Performance for Evaluating SQL Predicates

If the `WHERE` clause of your query includes comparisons with the operators `=`, `<=`, `<`, `>`, `>=`, `BETWEEN`, or `IN`, Kudu evaluates the condition directly and only returns the relevant results. This provides optimum performance, because Kudu only returns the relevant results to Impala.

For predicates such as `!=`, `LIKE`, or any other predicate type supported by Impala, Kudu does not evaluate the predicates directly. Instead, it returns all results to Impala and relies on Impala to evaluate the remaining predicates and filter the results accordingly. This may cause differences in performance, depending on the delta of the result set before and after evaluating the `WHERE` clause. In some cases, creating and periodically updating materialized views may be the right solution to work around these inefficiencies.

Inserting a Row

The syntax for inserting one or more rows using Impala is shown below.

```
INSERT INTO my_first_table VALUES (99, "sarah");  
INSERT INTO my_first_table VALUES (1, "john"), (2, "jane"), (3, "jim");
```

The primary key must not be null.

Inserting In Bulk

When inserting in bulk, there are at least three common choices. Each may have advantages and disadvantages, depending on your data and circumstances.

Multiple Single `INSERT` statements

This approach has the advantage of being easy to understand and implement. This approach is likely to be inefficient because Impala has a high query start-up cost compared to Kudu's insertion performance. This will lead to relatively high latency and poor throughput.

Single `INSERT` statement with multiple `VALUES` subclauses

If you include more than 1024 `VALUES` statements, Impala batches them into groups of 1024 (or the value of `batch_size`) before sending the requests to Kudu. This approach may perform slightly better than multiple sequential `INSERT` statements by amortizing the query start-up penalties on the Impala side. To set the batch size for the current Impala Shell session, use the following syntax:

```
set batch_size=10000;
```



Note: Increasing the Impala batch size causes Impala to use more memory. You should verify the impact on your cluster and tune accordingly.

Batch Insert

The approach that usually performs best, from the standpoint of both Impala and Kudu, is usually to import the data using a `SELECT FROM` subclause in Impala.

1. If your data is not already in Impala, one strategy is to [import it from a text file](#), such as a TSV or CSV file.
2. [Create the Kudu table](#), being mindful that the columns designated as primary keys cannot have null values.
3. Insert values into the Kudu table by querying the table containing the original data, as in the following example:

```
INSERT INTO my_kudu_table
SELECT * FROM legacy_data_import_table;
```

Ingest using the C++ or Java API

In many cases, the appropriate ingest path is to use the C++ or Java API to insert directly into Kudu tables. Unlike other Impala tables, data inserted into Kudu tables using the API becomes available for query in Impala without the need for any `INVALIDATE METADATA` statements or other statements needed for other Impala storage types.

INSERT and Primary Key Uniqueness Violations

In many relational databases, if you try to insert a row that has already been inserted, the insertion will fail because the primary key will be duplicated (see [Failures During INSERT, UPDATE, UPSERT, and DELETE Operations](#) on page 69). Impala, however, does not fail the query. Instead, it will generate a warning and continue to execute the remainder of the insert statement.

If you meant to replace existing rows from the table, use the `UPSERT` statement instead.

```
INSERT INTO my_first_table VALUES (99, "sarah");
UPSERT INTO my_first_table VALUES (99, "zoe");
```

The current value of the row is now `zoe`.

Updating a Row

The syntax for updating one or more rows using Impala is shown below.

```
UPDATE my_first_table SET name="bob" where id = 3;
```

You cannot change or null the primary key value.



Important: The `UPDATE` statement only works in Impala when the underlying data source is Kudu.

Updating In Bulk

You can update in bulk using the same approaches outlined in [Inserting In Bulk](#) on page 66.

Upserting a Row

The `UPSERT` command acts as a combination of the `INSERT` and `UPDATE` statements. For each row processed by the `UPSERT` statement:

- If another row already exists with the same set of primary key values, the other columns are updated to match the values from the row being 'UPSERTed'.
- If there is no row with the same set of primary key values, the row is created, the same as if the `INSERT` statement was used.

UPSERT Example

The following example demonstrates how the UPSERT statement works. We start by creating two tables, `foo1` and `foo2`.

```
CREATE TABLE foo1 (  
  id INT PRIMARY KEY,  
  col1 STRING,  
  col2 STRING  
)  
PARTITION BY HASH(id) PARTITIONS 3  
STORED AS KUDU;
```

```
CREATE TABLE foo2 (  
  id INT PRIMARY KEY,  
  col1 STRING,  
  col2 STRING  
)  
PARTITION BY HASH(id) PARTITIONS 3  
STORED AS KUDU;
```

Populate `foo1` and `foo2` using the following INSERT statements. For `foo2`, we leave column `col2` with NULL values to be upserted later:

```
INSERT INTO foo1 VALUES (1, "hi", "alice");
```

```
INSERT INTO foo2 select id, col1, NULL from foo1;
```

The contents of `foo2` will be:

```
SELECT * FROM foo2;  
...  
+----+-----+-----+  
| id | col1 | col2 |  
+----+-----+-----+  
| 1  | hi   | NULL |  
+----+-----+-----+  
Fetched 1 row(s) in 0.15s
```

Now use the UPSERT command to now replace the NULL values in `foo2` with the actual values from `foo1`.

```
UPSERT INTO foo2 (id, col2) select id, col2 from foo1;
```

```
SELECT * FROM foo2;  
...  
+----+-----+-----+  
| id | col1 | col2 |  
+----+-----+-----+  
| 1  | hi   | alice |  
+----+-----+-----+  
Fetched 1 row(s) in 0.15s
```

Altering a Table

You can use the ALTER TABLE statement to change the default value, encoding, compression, or block size of existing columns in a Kudu table.

The Impala SQL Reference [ALTER TABLE](#) includes a **Kudu Considerations** section with examples and a list of constraints relevant to altering a Kudu table in Impala.

Deleting a Row

You can delete Kudu rows in near real time using Impala.

```
DELETE FROM my_first_table WHERE id < 3;
```

You can even use more complex joins when deleting rows. For example, Impala uses a comma in the `FROM` sub-clause to specify a join query.

```
DELETE c FROM my_second_table c, stock_symbols s WHERE c.name = s.symbol;
```



Important: The `DELETE` statement only works in Impala when the underlying data source is Kudu.

Deleting In Bulk

You can delete in bulk using the same approaches outlined in [Inserting In Bulk](#) on page 66.

Failures During INSERT, UPDATE, UPSERT, and DELETE Operations

`INSERT`, `UPDATE`, and `DELETE` statements cannot be considered transactional as a whole. If one of these operations fails part of the way through, the keys may have already been created (in the case of `INSERT`) or the records may have already been modified or removed by another process (in the case of `UPDATE` or `DELETE`). You should design your application with this in mind.

Altering Table Properties

You can change Impala's metadata relating to a given Kudu table by altering the table's properties. These properties include the table name, the list of Kudu master addresses, and whether the table is managed by Impala (internal) or externally. You cannot modify a table's split rows after table creation.



Important: Altering table properties only changes Impala's metadata about the table, not the underlying table itself. These statements do not modify any Kudu data.

Rename an Impala Mapping Table

```
ALTER TABLE my_table RENAME TO my_new_table;
```

Renaming a table using the `ALTER TABLE ... RENAME` statement only renames the Impala mapping table, regardless of whether the table is an internal or external table. This avoids disruption to other applications that may be accessing the underlying Kudu table.

Rename the underlying Kudu table for an internal table

In CDH 5.14 and lower, if a table is an internal table, the underlying Kudu table may be renamed by changing the `kudu.table_name` property:

```
ALTER TABLE my_internal_table
SET TBLPROPERTIES('kudu.table_name' = 'new_name')
```

Remapping an external table to a different Kudu table

If another application has renamed a Kudu table under Impala, it is possible to re-map an external table to point to a different Kudu table name.

```
ALTER TABLE my_external_table_
SET TBLPROPERTIES('kudu.table_name' = 'some_other_kudu_table')
```

Change the Kudu Master Addresses

```
ALTER TABLE my_table SET TBLPROPERTIES('kudu.master_addresses' =  
'kudu-original-master.example.com:7051,kudu-new-master.example.com:7051');
```

Change an Internally-Managed Table to External

```
ALTER TABLE my_table SET TBLPROPERTIES('EXTERNAL' = 'TRUE');
```

Dropping a Kudu Table using Impala

If the table was created as an internal table in Impala, using `CREATE TABLE`, the standard `DROP TABLE` syntax drops the underlying Kudu table and all its data. If the table was created as an external table, using `CREATE EXTERNAL TABLE`, the mapping between Impala and Kudu is dropped, but the Kudu table is left intact, with all its data. To change an external table to internal, or vice versa, see [Altering Table Properties](#) on page 69.

```
DROP TABLE my_first_table;
```

Security Considerations

Kudu 1.3 (and higher) includes security features that allow Kudu clusters to be hardened against access from unauthorized users. Kudu uses strong authentication with Kerberos, while communication between Kudu clients and servers can now be encrypted with TLS. Kudu also allows you to use HTTPS encryption to connect to the web UI. These features should work seamlessly in Impala as long as Impala's user is given permission to access Kudu.

For instructions on how to configure a secure Kudu cluster, see [Kudu Security](#) on page 77.

Known Issues and Limitations

- When creating a Kudu table, the `CREATE TABLE` statement must include the primary key columns before other columns, in primary key order.
- Impala cannot update values in primary key columns.
- Impala cannot create Kudu tables with `VARCHAR` or nested-typed columns.
- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when used as an external table in Impala.
- Kudu tables with a column name containing upper case or non-ASCII characters cannot be used as an external table in Impala. Columns can be renamed in Kudu to work around this issue.
- `!=` and `LIKE` predicates are not pushed to Kudu, and instead will be evaluated by the Impala scan node. This may decrease performance relative to other types of predicates.
- Updates, inserts, and deletes using Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or use large tables.

Impala Keywords Not Supported for Creating Kudu Tables

- `PARTITIONED`
- `LOCATION`
- `ROWFORMAT`

Next Steps

The examples above have only explored a fraction of what you can do with Impala Shell.

- Learn about the [Impala project](#).
- Read the [Impala documentation](#).
- View the [Impala SQL Reference](#).
- For in-depth information on how to configure and use Impala to query Kudu data, see [Integrating Impala with Kudu](#).
- Read about Impala internals or learn how to contribute to Impala on the [Impala Wiki](#).

Using the Hive Metastore with Kudu

Kudu has an optional feature which allows it to integrate its own catalog with the Hive Metastore (HMS). The HMS is the de-facto standard catalog and metadata provider in the Hadoop ecosystem. When the HMS integration is enabled, Kudu tables can be discovered and used by external HMS-aware tools, even if they are not otherwise aware of, or integrated with Kudu. Additionally, these components can use the HMS to discover necessary information to connect to the Kudu cluster which owns the table, such as the Kudu master addresses.

Databases and Table Names

With the Hive Metastore integration disabled, Kudu presents tables as a single flat namespace, with no hierarchy or concept of a database. Additionally, Kudu's only restriction on table names is that they be a valid UTF-8 encoded string. When the HMS integration is enabled in Kudu, both of these properties change in order to match the HMS model: the table name must indicate the table's membership of a Hive database, and table name identifiers (i.e. the table name and database name) are subject to the Hive table name identifier constraints.

Databases

Hive has the concept of a database, which is a collection of individual tables. Each database forms its own independent namespace of table names.

In order to fit into this model, Kudu tables must be assigned a database when the HMS integration is enabled. No new APIs have been added to create or delete databases, nor are there APIs to assign an existing Kudu table to a database. Instead, a new convention has been introduced that Kudu table names must be in the format `<hive-database-name>.<hive-table-name>`. Thus, databases are an implicit part of the Kudu table name. By including databases as an implicit part of the Kudu table name, existing applications that use Kudu tables can operate on non-HMS-integrated and HMS-integrated table names with minimal or no changes.

Kudu provides no additional tooling to create or drop Hive databases. Administrators or users should use existing Hive tools such as the Beeline Shell or Impala to do so.

Naming Constraints

When the Hive Metastore integration is enabled, the database and table names of Kudu tables must follow the Hive Metastore naming constraints. Namely, the database and table name must contain only alphanumeric ASCII characters and underscores.



Note: When the `hive.support.special.characters.tablename` Hive configuration is `true`, the forward-slash (/) character in table name identifiers (i.e. the table name and database name) is also supported.

Additionally, the Hive Metastore does not enforce case sensitivity for table name identifiers. As such, when enabled, Kudu will follow suit and disallow tables from being created when one already exists whose table name identifier differs only by case. Operations that open, alter, or drop tables will also be case-insensitive for the table name identifiers.



Warning: Given the case insensitivity upon enabling the integration, if multiple Kudu tables exist whose names only differ by case, the Kudu master(s) will fail to start up. Be sure to rename such conflicting tables before enabling the Hive Metastore integration.

Metadata Synchronization

When the Hive Metastore integration is enabled, Kudu will automatically synchronize metadata changes to Kudu tables between Kudu and the HMS. As such, it is important to always ensure that the Kudu and HMS have a consistent view of existing tables, using the administrative tools described in the below section. Failure to do so may result in issues

like Kudu tables not being discoverable or usable by external, HMS-aware components (e.g. Apache Sentry, Apache Impala).



Note: The Hive Metastore automatically creates directories for Kudu tables. These directories are benign and can safely be ignored.

Impala has notions of internal and external Kudu tables. When dropping an internal table from Impala, the table's data is dropped in Kudu; in contrast when dropping an external table, the table's data is not dropped in Kudu. External tables may refer to tables by names that are different from the names of the underlying Kudu tables, while internal tables must use the same names as those stored in Kudu. Additionally, multiple external tables may refer to the same underlying Kudu table. Thus, since external tables may not map one-to-one with Kudu tables, the Hive Metastore integration and tooling will only automatically synchronize metadata for internal tables. See the [Kudu-Impala integration documentation](#) for more information about table types in Impala

Enabling the Hive Metastore Integration

Before enabling the Hive Metastore integration on an existing cluster, make sure to upgrade any tables that may exist in Kudu's or in the HMS's catalog. See the [Upgrading Existing Tables](#) on page 75 section for more details.

Setup using Cloudera Manager

1. When the Hive Metastore is configured with fine-grained authorization using Apache Sentry, and the Sentry HDFS Sync feature is enabled, the Kudu admin need to be able to access and modify directories that are created for Kudu by the HMS. This can be done by adding the Kudu admin user to the group of the Hive service users, e.g. by running the `usermod -aG hive kudu` command on the HMS nodes.
2. Go to the **Hive** service.
3. Click the **Configuration** tab.
4. Select the Kudu Service with which the Hive Metastore will synchronize the Kudu tables.

Manual Setup

1. When the Hive Metastore is configured with fine-grained authorization using Apache Sentry and the Sentry HDFS Sync feature is enabled, the Kudu admin needs to be able to access and modify directories that are created for Kudu by the HMS. This can be done by adding the Kudu admin user to the group of the Hive service users, e.g. by running the `usermod -aG hive kudu` command on the HMS nodes.
2. Configure the Hive Metastore to include the notification event listener and the Kudu HMS plugin, to allow altering and dropping columns, and to add full Thrift objects in notifications. Add the following values to the HMS configuration in `hive-site.xml`:

```
<property>
  <name>hive.metastore.transactional.event.listeners</name>
  <value>
    org.apache.hive.hcatalog.listener.DbNotificationListener,
    org.apache.kudu.hive.metastore.KuduMetastorePlugin
  </value>
</property>

<property>
  <name>hive.metastore.disallow.incompatible.col.type.changes</name>
  <value>>false</value>
</property>

<property>
  <name>hive.metastore.notifications.add.thrift.objects</name>
  <value>>true</value>
</property>
```

3. After building Kudu from source, add the `hms-plugin.jar` found under the build directory (e.g. `build/release/bin`) to the HMS classpath.

4. Restart the HMS.

5. Enable the Hive Metastore integration in Kudu with the following configuration properties for the Kudu master(s):

```
--hive_metastore_uris=<HMS Thrift URI(s)>  
--hive_metastore_sasl_enabled=<value of the Hive Metastore's hive.metastore.sasl.enabled  
configuration>
```



Note: In a secured cluster, in which `--hive_metastore_sasl_enabled` is set to true, `--hive_metastore_kerberos_principal` must match the primary portion of `hive.metastore.kerberos.principal` in the Hive Metastore configuration.

6. Restart the Kudu master(s).

Administrative Tools

Kudu provides the command line tools `kudu hms list`, `kudu hms precheck`, `kudu hms check`, and `kudu hms fix` to allow administrators to find and fix metadata inconsistencies between the internal Kudu catalog and the Hive Metastore catalog, during the upgrade process described below or during the normal operation of a Kudu cluster.

`kudu hms` tools should be run from the command line as the Kudu admin user. They require the full list of master addresses to be specified:

```
$ sudo -u kudu kudu hms check master-name-1:7051, master-name-2:7051, master-name-3:7051
```

To see a full list of the options available with the `kudu hms` tool, use the `--help` flag.



Note: When fine-grained authorization is enabled, the Kudu admin user, commonly "kudu", needs to have access to all the Kudu tables to be able to run the `kudu hms` tools. This can be done by configuring the user as a trusted user via the `--trusted_user_acl` master configuration. For more information, see [Trusted Users](#) on page 80.



Note: If the Hive Metastore is configured with fine-grained authorization using Apache Sentry, the Kudu admin user needs to have read and write privileges on HMS table entries. Configure this in the Hive Metastore using the `sentry.metastore.service.users` property.

```
kudu hms list
```

The `kudu hms list` tool scans the Hive Metastore catalog, and lists the HMS entries (including table name and type) for Kudu tables, as indicated by their HMS storage handler.

```
kudu hms check
```

The `kudu hms check` tool scans the Kudu and Hive Metastore catalogs, and validates that the two catalogs agree on what Kudu tables exist. The tool will make suggestions on how to fix any inconsistencies that are found. Typically, the suggestion will be to run the `kudu hms fix` tool, however some certain inconsistencies require using Impala Shell for fixing.

```
kudu hms precheck
```

The `kudu hms precheck` tool scans the Kudu catalog and validates that if there are multiple Kudu tables whose names only differ by case and logs the conflicted table names.

```
kudu hms fix
```

`kudu hms fix` tool analyzes the Kudu and HMS catalogs and attempts to fix any automatically-fixable issues, for instance, by creating a table entry in the HMS for each Kudu table that doesn't already have one. The `--dryrun` option shows the proposed fix instead of actually executing it. When no automatic fix is available, it will make suggestions on how a manual fix can help.



Note: The `kudu hms fix` tool will not automatically fix Impala external tables for the reasons described above. It is instead recommended to fix issues with external tables by dropping and recreating them.

```
kudu hms downgrade
```

The `kudu hms downgrade` downgrades the metadata to legacy format for Kudu and the Hive Metastores. It is discouraged to use unless necessary, since the legacy format can be deprecated in future releases.

Upgrading Existing Tables

Before enabling the Kudu-HMS integration, it is important to ensure that the Kudu and HMS start with a consistent view of existing tables. This may entail renaming Kudu tables to conform to the Hive naming constraints. This detailed workflow describes how to upgrade existing tables before enabling the Hive Metastore integration.

Prepare for the Upgrade

- Establish a maintenance window. During this time the Kudu cluster will still be available, but tables in Kudu and the Hive Metastore may be altered or renamed as a part of the upgrade process.
- Make note of all external tables using the following command and drop them. This reduces the chance of having naming conflicts with Kudu tables which can lead to errors during upgrading process. It also helps in cases where a catalog upgrade breaks external tables, due to the underlying Kudu tables being renamed. The external tables can be recreated after upgrade is complete.

```
$ sudo -u kudu kudu hms list master-name-1:7051, master-name-2:7051, master-name-3:7051
```

Perform the Upgrade

1. Run the `kudu hms precheck` tool to ensure no Kudu tables only differ by case. If the tool does not report any warnings, you can skip the next step.

```
$ sudo -u kudu kudu table rename_table
master-name-1:7051, master-name-2:7051, master-name-3:7051 <conflicting_table_name>
<new_table_name>
```

2. Run the `kudu hms check` tool using the following command. If the tool does not report any catalog inconsistencies, skip to Step 7 below.

```
$ sudo -u kudu kudu hms check master-name-1:7051, master-name-2:7051, master-name-3:7051
--hive_metastore_uris=<hive_metastore_uris>
[--ignore_other_clusters=<ignores_other_clusters>]
```



Note: By default, the `kudu hms` tools will ignore metadata in the HMS that refer to a different Kudu cluster than that being operated on, as indicated by having different masters specified. The tools compare the value of the `kudu.master_addresses` table property (either supplied at table creation or as `--kudu_master_hosts` on `impalad` daemons) in each HMS metadata entry against the RPC endpoints (including the ports) of the Kudu masters. To have the tooling account for and fix metadata entries with different master RPC endpoints specified (e.g. if ports are not specified in the HMS), supply `--ignore_other_clusters=false` as an argument to the `kudu hms check` and `fix` tools.

For example:

```
$ sudo -u kudu kudu hms check master-name-1:7051,master-name-2:7051,master-name-3:7051
--hive_metastore_uris=thrift://hive-metastore:9083 --ignore_other_clusters=false
```

3. If the `kudu hms check` tool reports an inconsistent catalog, perform a dry-run of the `kudu hms fix` tool to understand how the tool will attempt to address the automatically-fixable issues.

```
$ sudo -u kudu kudu hms fix master-name-1:7051,master-name-2:7051,master-name-3:7051
--hive_metastore_uris=<hive_metastore_uris> --dryrun=true
[--ignore_other_clusters=<ignore_other_clusters>]
```

For example:

```
$ sudo -u kudu kudu hms check master-name-1:7051,master-name-2:7051,master-name-3:7051
--hive_metastore_uris=thrift://hive-metastore:9083 --dryrun=true
--ignore_other_clusters=false
```

4. Manually fix any issues that are reported by the check tool that cannot be automatically fixed. For example, rename any tables with names that are not Hive-conformant.
5. Run the `kudu hms fix` tool to automatically fix all the remaining issues.

```
$ sudo -u kudu kudu hms fix master-name-1:7051,master-name-2:7051,master-name-3:7051
--hive_metastore_uris=<hive_metastore_uris>
[--drop_orphan_hms_tables=<drop_orphan_hms_tables>]
[--ignore_other_clusters=<ignore_other_clusters>]
```

For example:

```
$ sudo -u kudu kudu hms fix master-name-1:7051,master-name-2:7051,master-name-3:7051
--hive_metastore_uris=thrift://hive-metastore:9083 --ignore_other_clusters=false
```



Note: The `--drop_orphan_hms_tables` argument indicates whether to drop orphan Hive Metastore tables that refer to non-existent Kudu tables. Due to [KUDU-2883](#), this option may fail to drop HMS entries that have no table ID. A workaround to this is to drop the table via the Impala Shell.

6. Recreate any external tables that were dropped when preparing for the upgrade by using Impala Shell.
7. Enable the Hive Metastore Integration as described in the [Enabling the Hive Metastore Integration](#) on page 73 section.

Kudu Security

Kudu includes security features that allow Kudu clusters to be hardened against access from unauthorized users. Kudu uses strong authentication with Kerberos, while communication between Kudu clients and servers can now be encrypted with TLS. Kudu also allows you to use HTTPS encryption to connect to the web UI.

The rest of this topic describes the security capabilities of Apache Kudu and how to configure a secure Kudu cluster. Currently, there are a few known limitations in Kudu security that might impact your cluster. For the list, see [Security Limitations](#) on page 18.

Kudu Authentication with Kerberos

Kudu can be configured to enforce secure authentication among servers, and between clients and servers. Authentication prevents untrusted actors from gaining access to Kudu, and securely identifies connecting users or services for authorization checks. Authentication in Kudu is designed to interoperate with other secure Hadoop components by utilizing Kerberos.

Configure authentication on Kudu servers using the `--rpc_authentication` flag, which can be set to one of the following options:

- `required` - Kudu will reject connections from clients and servers who lack authentication credentials.
- `optional` - Kudu will attempt to use strong authentication, but will allow unauthenticated connections.
- `disabled` - Kudu will only allow unauthenticated connections.

By default, the flag is set to `optional`. To secure your cluster, set `--rpc_authentication` to `required`.

Internal Private Key Infrastructure (PKI)

Kudu uses an internal PKI to issue X.509 certificates to servers in the cluster. Connections between peers who have both obtained certificates will use TLS for authentication. In such cases, neither peer needs to contact the Kerberos KDC.

X.509 certificates are only used for internal communication among Kudu servers, and between Kudu clients and servers. These certificates are never presented in a public facing protocol. By using internally-issued certificates, Kudu offers strong authentication which scales to huge clusters, and allows TLS encryption to be used without requiring you to manually deploy certificates on every node.

Authentication Tokens

After authenticating to a secure cluster, the Kudu client will automatically request an authentication token from the Kudu master. An authentication token encapsulates the identity of the authenticated user and carries the Kudu master's RSA signature so that its authenticity can be verified. This token will be used to authenticate subsequent connections. By default, authentication tokens are only valid for seven days, so that even if a token were compromised, it cannot be used indefinitely. For the most part, authentication tokens should be completely transparent to users. By using authentication tokens, Kudu is able to take advantage of strong authentication, without paying the scalability cost of communicating with a central authority for every connection.

When used with distributed compute frameworks such as Apache Spark, authentication tokens can simplify configuration and improve security. For example, the Kudu Spark connector will automatically retrieve an authentication token during the planning stage, and distribute the token to tasks. This allows Spark to work against a secure Kudu cluster where only the planner node has Kerberos credentials.

Client Authentication to Secure Kudu Clusters

Users running client Kudu applications must first run the `kinit` command to obtain a Kerberos ticket-granting ticket. For example:

```
kinit admin@EXAMPLE-REALM.COM
```

Once authenticated, you use the same client code to read from and write to Kudu servers with and without the Kerberos configuration.

Scalability

Kudu authentication is designed to scale to thousands of nodes, which means it must avoid unnecessary coordination with a central authentication authority (such as the Kerberos KDC) for each connection. Instead, Kudu servers and clients use Kerberos to establish initial trust with the Kudu master, and then use alternate credentials for subsequent connections. As described previously, the Kudu master issues internal X.509 certificates to tablet servers on startup, and temporary authentication tokens to clients on first contact.

Coarse-grained Authorization

Kudu supports coarse-grained authorization checks for client requests based on the client's authenticated Kerberos principal (user or service). Access levels are granted based on whitelist-style Access Control Lists (ACLs), one for each level. Each ACL specifies a comma-separated list of users, or may be set to '*' to indicate that all authenticated users have access rights at the specified level.

The two levels of access which can be configured are:

- **Superuser** - Principals authorized as a superuser can perform certain administrative functions such as using the `kudu` command line tool to diagnose and repair cluster issues.
- **User** - Principals authorized as a user are able to access and modify all data in the Kudu cluster. This includes the ability to create, drop, and alter tables, as well as read, insert, update, and delete data. The default value for the User ACL is '*', which allows all users access to the cluster. However, if authentication is enabled, this will restrict access to only those users who are able to successfully authenticate using Kerberos. Unauthenticated users on the same network as the Kudu servers will be unable to access the cluster.



Note: Internally, Kudu has a third access level for the daemons themselves called **Service**. This is used to ensure that users cannot connect to the cluster and pose as tablet servers.

Fine-Grained Authorization

As of Kudu 1.10.0, Kudu can be configured to enforce fine-grained authorization across servers. This ensures that users can see only the data they are explicitly authorized to see. Kudu currently supports this by leveraging policies defined in Apache Sentry 2.2 and later.



Warning: Fine-grained authorization policies are not enforced when accessing the web UI. User data may appear on various pages of the web UI (e.g. in logs, metrics, scans, etc.). As such, it is recommended to either limit access to the web UI ports, or redact or disable the web UI entirely, as desired. See the [Web UI Encryption](#) on page 82 section for information on securing the web UI.

Apache Sentry

Apache Sentry models tabular objects in the following hierarchy:

- **Server:** is indicated by the Kudu configuration flag `--server_name`. Everything stored in a Kudu cluster falls within the given "server"
- **Database:** is indicated as a prefix of table names with the format `<database>.<table>`
- **Table:** is a single Kudu table.
- **Column:** is a column within a Kudu table.

Each level of this hierarchy defines a scope on which the privileges can be granted. Privileges granted on a higher scope imply privileges on a lower scope. For example, if a user has a `SELECT` privilege on a database, then that user implicitly has the `SELECT` privileges on every table belonging to that database.

Privileges are also associated with specific actions. Access to Kudu tables may rely on privileges on the following actions:

- ALTER
- CREATE
- DELETE
- DROP
- INSERT
- UPDATE
- SELECT

Additionally, there are three special actions recognized by Kudu: `ALL`, `OWNER`, and `METADATA`. If a user has the `ALL` or `OWNER` privileges on a given table, that user has all of the above privileges on the table. `METADATA` privilege is not an actual privilege per se, rather, it is a conceptual privilege with which Kudu models any privilege. If a user has any privilege on a given table, that user has `METADATA` privileges on the table, i.e. a privilege granted on any action on a table implies that the user has the `METADATA` privilege on that table.

For more details about Sentry privileges, see the [Apache Sentry documentation](#).



Note: Depending on the value of the `sentry.db.explicit.grants.permitted` configuration in Sentry, certain privileges may not be grantable in Sentry. For example, in Sentry deployments that don't support `UPDATE` privileges, to perform an operation that requires `UPDATE` privileges, a user must instead have `ALL` privileges.

When a Kudu master receives a request, it consults Sentry to determine what privileges a user has. If the user is not authorized to perform the requested action, the request is rejected. Kudu leverages the authenticated identity of a user to decide whether to perform or reject a request.

Authorization Tokens

Rather than having every tablet server communicate directly with Sentry, privileges are propagated and checked via **authorization tokens**. These tokens encapsulate what privileges a user has on a given table. Tokens are generated by the master and returned to Kudu clients upon opening a Kudu table. Kudu clients automatically attach authorization tokens when sending requests to tablet servers.

Authorization tokens are a means to limit the number of nodes directly accessing Sentry to retrieve privileges. As such, since the expected number of tablet servers in a cluster is much higher than the number of Kudu masters, they are only used to authorize requests sent to tablet servers. Kudu masters fetch privileges directly from Sentry or cache. See <<privilege-caching>> for more details of Kudu's privilege cache.

Similar to the validity interval for authentication tokens, to limit the window of potential unwanted access if a token becomes compromised, authorization tokens are valid for five minutes by default. The acquisition and renewal of a token is hidden from the user, as Kudu clients automatically retrieve new tokens when existing tokens expire.

When a tablet server that has been configured to enforce fine-grained access control receives a request, it checks the privileges in the attached token, rejecting it if the privileges are not sufficient to perform the requested operation, or if it is invalid (e.g. expired).

Trusted Users

It may be desirable to allow certain users to view and modify any data stored in Kudu. Such users can be specified via the `--trusted_user_acl` master configuration. Trusted users can perform any operation that would otherwise require fine-grained privileges, without Kudu consulting Sentry.

Additionally, some services that interact with Kudu may authorize requests on behalf of their end users. For example, Apache Impala authorizes queries on behalf of its users, and sends requests to Kudu as the Impala service user, commonly "impala". Since Impala authorizes requests on its own, to avoid extraneous communication between Sentry and Kudu, the Impala service user should be listed as a trusted user.



Note: When accessing Kudu through Impala, Impala enforces its own fine-grained authorization policy. This policy is similar to Kudu's and can be found in [Impala's authorization documentation](#).

Configuring the Integration with Apache Sentry

Sentry is often configured with Kerberos authentication. In order to enable integration with Sentry, a cluster must first be integrated with the Apache Hive Metastore. See the [Enabling the Hive Metastore Integration](#) on page 73 section to configure Kudu to synchronize its internal catalog with the Hive Metastore.

The following configurations must be set on the master:

```
--sentry_service_rpc_addresses=<Sentry RPC address>
--server_name=<value of HiveServer2's hive.sentry.server configuration>
--kudu_service_name=kudu
--sentry_service_kerberos_principal=sentry
--sentry_service_security_mode=kerberos

# This example ACL setup allows the 'impala' user to access all data stored in
# Kudu, assuming Impala will authorize requests on its own. The 'hadoopadmin'
# user is also granted access to all Kudu data, which may facilitate testing
# and debugging.
--trusted_user_acl=impala,hadoopadmin
```

The following configurations must be set on the tablet servers:

```
--tserver_enforce_access_control=true
```

Caching

To avoid overwhelming Sentry with requests to fetch user privileges, the Kudu master can be configured to cache user privileges. A by-product of this caching is that when privileges are changed in Sentry, they may not be reflected in Kudu for a configurable amount of time, defined by the following Kudu master configurations:

```
--sentry_privileges_cache_ttl_factor * --authz_token_validity_interval_secs
```

The default value is fifty minutes. If privilege updates need to be reflected in Kudu sooner than this, the Kudu CLI tool can be used to invalidate the cached privileges to force Kudu to fetch new ones from Sentry:

```
kudu master authz_cache reset <master-addresses>
```

Policy for Kudu Masters

The following authorization policy is enforced by Kudu masters:

Table 5: Authorization Policy for Masters

Operation	Required Privilege
CreateTable	CREATE ON DATABASE

Operation	Required Privilege
CreateTable with a different owner specified than the requesting user	ALL ON DATABASE with the Sentry GRANT OPTION. See GRANT <Privilege> ... WITH GRANT OPTION .
DeleteTable	DROP ON TABLE
AlterTable (with no rename)	ALTER ON TABLE
AlterTable (with rename)	ALL ON TABLE <old-table> and CREATE ON DATABASE <new-database>
IsCreateTableDone	METADATA ON TABLE
IsAlterTableDone	METADATA ON TABLE
ListTables	METADATA ON TABLE
GetTableLocations	METADATA ON TABLE
GetTableSchema	METADATA ON TABLE
GetTabletLocations	METADATA ON TABLE

Policy for Kudu Tablet Servers

The following authorization policy is enforced by Kudu tablet servers:

Table 6: Authorization Policy for Tablet Servers

Operation	Required Privilege
Scan	SELECT ON TABLE, or METADATA ON TABLE and SELECT ON COLUMN for each projected column and each predicate column
Scan (no projected columns, equivalent to COUNT(*))	SELECT ON TABLE, or SELECT ON COLUMN for each column in the table
Scan (with virtual columns)	SELECT ON TABLE, or SELECT ON COLUMN for each column in the table
Scan (in ORDERED mode)	<privileges required for a Scan> and SELECT ON COLUMN for each primary key column
Insert	INSERT ON TABLE
Update	UPDATE ON TABLE
Upsert	INSERT ON TABLE and UPDATE ON TABLE
Delete	DELETE ON TABLE
SplitKeyRange	SELECT ON COLUMN for each primary key column and SELECT ON COLUMN for each projected column
Checksum	User must be configured in --superuser_acl
ListTablets	User must be configured in --superuser_acl



Note: Unlike Impala, Kudu only supports all-or-nothing access to a table's schema, rather than showing only authorized columns.

Encryption

Kudu allows you to use TLS to encrypt all communications among servers, and between clients and servers. Configure TLS encryption on Kudu servers using the `--rpc_encryption` flag, which can be set to one of the following options:

- `required` - Kudu will reject unencrypted connections.
- `optional` - Kudu will attempt to use encryption, but will allow unencrypted connections.
- `disabled` - Kudu will not use encryption.

By default, the flag is set to `optional`. To secure your cluster, set `--rpc_encryption` to `required`.



Note: Kudu will automatically turn off encryption on local loopback connections, since traffic from these connections is never exposed externally. This allows locality-aware compute frameworks, such as Spark and Impala, to avoid encryption overhead, while still ensuring data confidentiality.

Using Vormetric encryption is considered experimental. We recommend you to experiment using Vormetric encryption with Kudu in a development environment.

Web UI Encryption

The Kudu web UI can be configured to use secure HTTPS encryption by providing each server with TLS certificates. Use the `--webserver_certificate_file` and `--webserver_private_key_file` properties to specify the certificate and private key to be used for communication.

Alternatively, you can choose to completely disable the web UI by setting `--webserver_enabled` flag to `false` on the Kudu servers.

Web UI Redaction

To prevent sensitive data from being included in the web UI, all row data is redacted. Table metadata, such as table names, column names, and partitioning information is not redacted. Alternatively, you can choose to completely disable the web UI by setting the `--webserver_enabled` flag to `false` on the Kudu servers.



Note: Disabling the web UI will also disable REST endpoints such as `/metrics`. Monitoring systems rely on these endpoints to gather metrics data.

Log Redaction

To prevent sensitive data from being included in Kudu server logs, all row data will be redacted. You can turn off log redaction using the `--redact` flag.

Configuring a Secure Kudu Cluster using Cloudera Manager



Warning: If you are upgrading from Kudu 1.2.0 / CDH 5.10.x, you must upgrade both Kudu and CDH parcels (or packages) at the same time. If you upgrade Kudu but do not upgrade CDH, new Kudu features such as Security will not be available. Note that even though you might be able to see the updated configuration options for Kudu security in Cloudera Manager, configuring them will have no effect.

Use the following set of instructions to secure a Kudu cluster using Cloudera Manager:

Enabling Kerberos Authentication and RPC Encryption



Important: The following instructions assume you already have a secure Cloudera Manager cluster with Kerberos authentication enabled. If this is not the case, first secure your cluster using the steps described at [Enabling Kerberos Authentication Using the Cloudera Manager Wizard](#).

To enable Kerberos authentication for Kudu:

1. Go to the **Kudu** service.
2. Click the **Configuration** tab.
3. Select **Category > Main**.
4. In the Search field, type **Kerberos** to show the relevant properties.
5. Edit the following properties according to your cluster configuration:

Field	Usage Notes
Kerberos Principal	Set to the default principal, <code>kudu</code> . Currently, Kudu does not support configuring a custom service principal for Kudu processes.
Enable Secure Authentication And Encryption	Select this checkbox to enable authentication and RPC encryption between all Kudu clients and servers, as well as between individual servers. Only enable this property after you have configured Kerberos.

6. Click **Save Changes**.
7. You will see an error message that tells you the Kudu keytab is missing. To generate the keytab, go to the top navigation bar and click **Administration > Security**.
8. Go to the **Kerberos Credentials** tab. On this page you will see a list of the existing Kerberos principals for services running on the cluster.
9. Click **Generate Missing Credentials**. Once the Generate Missing Credentials command has finished running, you will see the Kudu principal added to the list.

Configuring Coarse-grained Authorization with ACLs

1. Go to the **Kudu** service.
2. Click the **Configuration** tab.
3. Select **Category > Security**.
4. In the Search field, type **ACL** to show the relevant properties.
5. Edit the following properties according to your cluster configuration:

Field	Usage Notes
Superuser Access Control List	Add a comma-separated list of superusers who can access the cluster. By default, this property is left blank. '*' indicates that all authenticated users will be given superuser access.

Field	Usage Notes
User Access Control List	<p>Add a comma-separated list of users who can access the cluster. By default, this property is set to '*'.</p> <p>The default value of '*' allows all users access to the cluster. However, if authentication is enabled, this will restrict access to only those users who are able to successfully authenticate using Kerberos. Unauthenticated users on the same network as the Kudu servers will be unable to access the cluster.</p> <p>Add the <code>impala</code> user to this list to allow Impala to query data in Kudu. You might choose to add any other relevant usernames if you want to give access to Spark Streaming jobs.</p>

6. Click **Save Changes**.

Configuring HTTPS Encryption for the Kudu Master and Tablet Server Web UIs

Use the following steps to enable HTTPS for encrypted connections to the Kudu master and tablet server web UIs.

1. Go to the **Kudu** service.
2. Click the **Configuration** tab.
3. Select **Category > Security**.
4. In the Search field, type **TLS/SSL** to show the relevant properties.
5. Edit the following properties according to your cluster configuration:

Field	Usage Notes
Master TLS/SSL Server Private Key File (PEM Format)	Set to the path containing the Kudu master host's private key (PEM-format). This is used to enable TLS/SSL encryption (over HTTPS) for browser-based connections to the Kudu master web UI.
Tablet Server TLS/SSL Server Private Key File (PEM Format)	Set to the path containing the Kudu tablet server host's private key (PEM-format). This is used to enable TLS/SSL encryption (over HTTPS) for browser-based connections to Kudu tablet server web UIs.
Master TLS/SSL Server Certificate File (PEM Format)	Set to the path containing the signed certificate (PEM-format) for the Kudu master host's private key (set in Master TLS/SSL Server Private Key File). The certificate file can be created by concatenating all the appropriate root and intermediate certificates required to verify trust.
Tablet Server TLS/SSL Server Certificate File (PEM Format)	Set to the path containing the signed certificate (PEM-format) for the Kudu tablet server host's private key (set in Tablet Server TLS/SSL Server Private Key File). The certificate file can be created by concatenating all the appropriate root and intermediate certificates required to verify trust.
Enable TLS/SSL for Master Server	Enables HTTPS encryption on the Kudu master web UI.
Enable TLS/SSL for Tablet Server	Enables HTTPS encryption on the Kudu tablet server Web UIs.

6. Click **Save Changes**.

Enabling Sentry Authorization

To enable Kudu's integration with Sentry:

1. First, ensure that Kudu has been configured to synchronize its catalog with the Hive Metastore. See the steps described in [Enabling the Hive Metastore Integration](#) on page 73.
2. Go to the **Kudu** service.

3. Click the **Configuration** tab.
4. Select the Sentry Service with which Kudu should authorize requests.

Configuring a Secure Kudu Cluster using the Command Line



Important: Follow these command-line instructions on systems that do not use Cloudera Manager. If you are using Cloudera Manager, see [Configuring a Secure Kudu Cluster using Cloudera Manager](#) on page 83.

The following configuration parameters should be set on all servers (master and tablet servers) to ensure that a Kudu cluster is secure:

```
# Connection Security
#-----
--rpc_authentication=required
--rpc_encryption=required
--keytab_file=<path-to-kerberos-keytab>

# Web UI Security
#-----
--webserver_certificate_file=<path-to-cert-pem>
--webserver_private_key_file=<path-to-key-pem>
# optional
--webserver_private_key_password_cmd=<password-cmd>

# If you prefer to disable the web UI entirely:
--webserver_enabled=false

# Coarse-grained authorization
#-----

# This example ACL setup allows the 'impala' user as well as the
# 'etl_service_account' principal access to all data in the
# Kudu cluster. The 'hadoopadmin' user is allowed to use administrative
# tooling. Note that by granting access to 'impala', other users
# may access data in Kudu via the Impala service subject to its own
# authorization rules.
--user_acl=impala,etl_service_account
--admin_acl=hadoopadmin
```

More information about these flags can be found in the [configuration reference documentation](#).

See [Configuring the Integration with Apache Sentry](#) on page 80 to see an example of how to enable fine-grained authorization via Apache Sentry.

Apache Kudu Schema Design

Kudu tables have a structured data model similar to tables in a traditional relational database. With Kudu, schema design is critical for achieving the best performance and operational stability. Every workload is unique, and there is no single schema design that is best for every table. This topic outlines effective schema design philosophies for Kudu, and how they differ from approaches used for traditional relational database schemas.

There are three main concerns when creating Kudu tables: column design, primary key design, and partitioning.

The Perfect Schema

The perfect schema would accomplish the following:

- Data would be distributed such that reads and writes are spread evenly across tablet servers. This can be achieved by effective partitioning.
- Tablets would grow at an even, predictable rate, and load across tablets would remain steady over time. This can be achieved by effective partitioning.
- Scans would read the minimum amount of data necessary to fulfill a query. This is impacted mostly by primary key design, but partitioning also plays a role via partition pruning.

The perfect schema depends on the characteristics of your data, what you need to do with it, and the topology of your cluster. Schema design is the single most important thing within your control to maximize the performance of your Kudu cluster.

Column Design

A Kudu table consists of one or more columns, each with a defined type. Columns that are not part of the primary key may be nullable. Supported column types include:

- boolean
- 8-bit signed integer
- 16-bit signed integer
- 32-bit signed integer
- 64-bit signed integer
- `unixtime_micros` (64-bit microseconds since the Unix epoch)
- single-precision (32-bit) IEEE-754 floating-point number
- double-precision (64-bit) IEEE-754 floating-point number
- decimal (see [Decimal Type](#) on page 87 for details)
- UTF-8 encoded string (up to 64KB uncompressed)
- binary (up to 64KB uncompressed)

Kudu takes advantage of strongly-typed columns and a columnar on-disk storage format to provide efficient encoding and serialization. To make the most of these features, columns should be specified as the appropriate type, rather than simulating a 'schemaless' table using string or binary columns for data which could otherwise be structured. In addition to encoding, Kudu allows compression to be specified on a per-column basis.

Decimal Type

The `decimal` type is a numeric data type with fixed scale and precision suitable for financial and other arithmetic calculations where the imprecise representation and rounding behavior of `float` and `double` make those types impractical. The `decimal` type is also useful for integers larger than `int64` and cases with fractional values in a primary key.

The `decimal` type is a parameterized type that takes precision and scale type attributes.

Precision represents the total number of digits that can be represented by the column, regardless of the location of the decimal point. This value must be between 1 and 38 and has no default. For example, a precision of 4 is required to represent integer values up to 9999, or to represent values up to 99.99 with two fractional digits. You can also represent corresponding negative values, without any change in the precision. For example, the range -9999 to 9999 still only requires a precision of 4.

Scale represents the number of fractional digits. This value must be between 0 and the precision. A scale of 0 produces integral values, with no fractional part. If precision and scale are equal, all of the digits come after the decimal point. For example, a decimal with precision and scale equal to 3 can represent values between -0.999 and 0.999.

Performance considerations:

Kudu stores each value in as few bytes as possible depending on the precision specified for the decimal column. For that reason it is not advised to just use the highest precision possible for convenience. Doing so could negatively impact performance, memory and storage.

Before encoding and compression:

- Decimal values with precision of 9 or less are stored in 4 bytes.
- Decimal values with precision of 10 through 18 are stored in 8 bytes.
- Decimal values with precision greater than 18 are stored in 16 bytes.



Note: The precision and scale of decimal columns cannot be changed by altering the table.

Column Encoding

Depending on the type of the column, Kudu columns can be created with the following encoding types.

Plain Encoding

Data is stored in its natural format. For example, `int32` values are stored as fixed-size 32-bit little-endian integers.

Bitshuffle Encoding

A block of values is rearranged to store the most significant bit of every value, followed by the second most significant bit of every value, and so on. Finally, the result is LZ4 compressed. Bitshuffle encoding is a good choice for columns that have many repeated values, or values that change by small amounts when sorted by primary key. The bitshuffle project has a good overview of performance and use cases.

Run Length Encoding

Runs (consecutive repeated values) are compressed in a column by storing only the value and the count. Run length encoding is effective for columns with many consecutive repeated values when sorted by primary key.

Dictionary Encoding

Dictionary encoding can be used for `BINARY` or `STRING` columns. A dictionary of unique values is built, and each column value is encoded as its corresponding index in the dictionary. Dictionary encoding is effective for columns with low cardinality. If the column values of a given row set are unable to be compressed because the number of unique values is too high, Kudu will transparently fall back to plain encoding for that row set. This is evaluated during flush.

Prefix Encoding

Common prefixes are compressed in consecutive column values. Prefix encoding can be effective for values that share common prefixes, or the first column of the primary key, since rows are sorted by primary key within tablets.

Each column in a Kudu table can be created with an encoding, based on the type of the column. Starting with Kudu 1.3, default encodings are specific to each column type.

Column Type	Encoding	Default
int8, int16, int32	plain, bitshuffle, run length	bitshuffle
int64, unixtime_micros	plain, bitshuffle, run length	bitshuffle
float, double	plain, bitshuffle	bitshuffle
bool	plain, run length	run length
string, binary	plain, prefix, dictionary	dictionary

Column Compression

Kudu allows per-column compression using the LZ4, Snappy, or zlib compression codecs.

By default, columns that are Bitshuffle-encoded are inherently compressed with the LZ4 compression. Otherwise, columns are stored uncompressed. Consider using compression if reducing storage space is more important than raw scan performance.

Every data set will compress differently, but in general LZ4 is the most efficient codec, while zlib will compress to the smallest data sizes. Bitshuffle-encoded columns are automatically compressed using LZ4, so it is not recommended to apply additional compression on top of this encoding.

Primary Key Design

Every Kudu table must declare a primary key comprised of one or more columns. Like an RDBMS primary key, the Kudu primary key enforces a uniqueness constraint. Attempting to insert a row with the same primary key values as an existing row will result in a duplicate key error.

Primary key columns must be non-nullable, and may not be a boolean or floating-point type.

Once set during table creation, the set of columns in the primary key may not be altered.

Unlike an RDBMS, Kudu does not provide an auto-incrementing column feature, so the application must always provide the full primary key during insert.

Row delete and update operations must also specify the full primary key of the row to be changed. Kudu does not natively support range deletes or updates.

The primary key values of a column may not be updated after the row is inserted. However, the row may be deleted and re-inserted with the updated value.

Primary Key Index

As with many traditional relational databases, Kudu's primary key is in a clustered index. All rows within a tablet are sorted by its primary key.

When scanning Kudu rows, use equality or range predicates on primary key columns to efficiently find the rows.

Considerations for Backfill Inserts

This section discusses a primary key design consideration for timeseries use cases where the primary key is a timestamp, or the first column of the primary key is a timestamp.

Each time a row is inserted into a Kudu table, Kudu looks up the primary key in the primary key index storage to check whether that primary key is already present in the table. If the primary key exists in the table, a "duplicate key" error is returned. In the typical case where data is being inserted at the current time as it arrives from the data source, only a small range of primary keys are "hot". So, each of these "check for presence" operations is very fast. It hits the cached primary key storage in memory and doesn't require going to disk.

In the case when you load historical data, which is called "backfilling", from an offline data source, each row that is inserted is likely to hit a cold area of the primary key index which is not resident in memory and will cause one or more HDD disk seeks. For example, in a normal ingestion case where Kudu sustains a few million inserts per second, the "backfill" use case might sustain only a few thousand inserts per second.

To alleviate the performance issue during backfilling, consider the following options:

- Make the primary keys more compressible.
For example, with the first column of a primary key being a random ID of 32-bytes, caching one billion primary keys would require at least 32 GB of RAM to stay in cache. If caching backfill primary keys from several days ago, you need to have several times 32 GB of memory. By changing the primary key to be more compressible, you increase the likelihood that the primary keys can fit in cache and thus reducing the amount of random disk I/Os.
- Use SSDs for storage as random seeks are orders of magnitude faster than spinning disks.
- Change the primary key structure such that the backfill writes hit a continuous range of primary keys.

Partitioning

In order to provide scalability, Kudu tables are partitioned into units called tablets, and distributed across many tablet servers. A row always belongs to a single tablet. The method of assigning rows to tablets is determined by the partitioning of the table, which is set during table creation.

Choosing a partitioning strategy requires understanding the data model and the expected workload of a table. For write-heavy workloads, it is important to design the partitioning such that writes are spread across tablets in order to avoid overloading a single tablet. For workloads involving many short scans, where the overhead of contacting remote servers dominates, performance can be improved if all of the data for the scan is located on the same tablet. Understanding these fundamental trade-offs is central to designing an effective partition schema.



Important: Kudu does not provide a default partitioning strategy when creating tables. It is recommended that new tables which are expected to have heavy read and write workloads have at least as many tablets as tablet servers.

Kudu provides two types of partitioning: range partitioning and hash partitioning. Tables may also have multilevel partitioning, which combines range and hash partitioning, or multiple instances of hash partitioning.

Range Partitioning

Range partitioning distributes rows using a totally-ordered range partition key. Each partition is assigned a contiguous segment of the range partition key space. The key must be comprised of a subset of the primary key columns. If the range partition columns match the primary key columns, then the range partition key of a row will equal its primary key. In range partitioned tables without hash partitioning, each range partition will correspond to exactly one tablet.

The initial set of range partitions is specified during table creation as a set of partition bounds and split rows. For each bound, a range partition will be created in the table. Each split will divide a range partition in two. If no partition bounds are specified, then the table will default to a single partition covering the entire key space (unbounded below and above). Range partitions must always be non-overlapping, and split rows must fall within a range partition.

Adding and Removing Range Partitions

Kudu allows range partitions to be dynamically added and removed from a table at runtime, without affecting the availability of other partitions. Removing a partition will delete the tablets belonging to the partition, as well as the data contained in them. Subsequent inserts into the dropped partition will fail. New partitions can be added, but they

must not overlap with any existing range partitions. Kudu allows dropping and adding any number of range partitions in a single transactional alter table operation.

Dynamically adding and dropping range partitions is particularly useful for time series use cases. As time goes on, range partitions can be added to cover upcoming time ranges. For example, a table storing an event log could add a month-wide partition just before the start of each month in order to hold the upcoming events. Old range partitions can be dropped in order to efficiently remove historical data, as necessary.

Hash Partitioning

Hash partitioning distributes rows by hash value into one of many buckets. In single-level hash partitioned tables, each bucket will correspond to exactly one tablet. The number of buckets is set during table creation. Typically the primary key columns are used as the columns to hash, but as with range partitioning, any subset of the primary key columns can be used.

Hash partitioning is an effective strategy when ordered access to the table is not needed. Hash partitioning is effective for spreading writes randomly among tablets, which helps mitigate hot-spotting and uneven tablet sizes.

Multilevel Partitioning

Kudu allows a table to combine multiple levels of partitioning on a single table. Zero or more hash partition levels can be combined with an optional range partition level. The only additional constraint on multilevel partitioning beyond the constraints of the individual partition types, is that multiple levels of hash partitions must not hash the same columns.

When used correctly, multilevel partitioning can retain the benefits of the individual partitioning types, while reducing the downsides of each. The total number of tablets in a multilevel partitioned table is the product of the number of partitions in each level.

Partition Pruning

Kudu scans will automatically skip scanning entire partitions when it can be determined that the partition can be entirely filtered by the scan predicates. To prune hash partitions, the scan must include equality predicates on every hashed column. To prune range partitions, the scan must include equality or range predicates on the range partitioned columns. Scans on multilevel partitioned tables can take advantage of partition pruning on any of the levels independently.

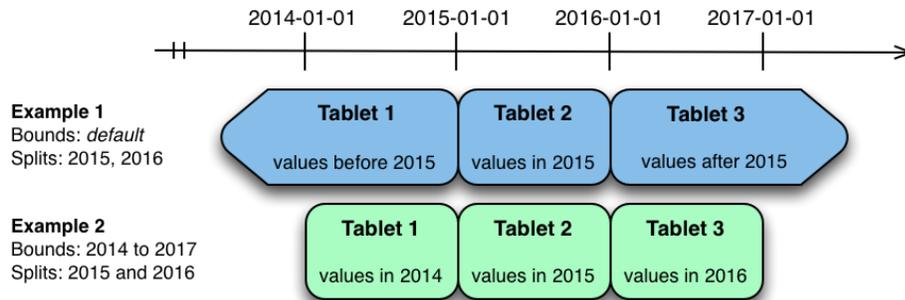
Partitioning Examples

To illustrate the factors and tradeoffs associated with designing a partitioning strategy for a table, we will walk through some different partitioning scenarios. Consider the following table schema for storing machine metrics data (using SQL syntax and date-formatted timestamps for clarity):

```
CREATE TABLE metrics (  
  host STRING NOT NULL,  
  metric STRING NOT NULL,  
  time INT64 NOT NULL,  
  value DOUBLE NOT NULL,  
  PRIMARY KEY (host, metric, time)  
);
```

Range Partitioning

A natural way to partition the `metrics` table is to range partition on the `time` column. Let's assume that we want to have a partition per year, and the table will hold data for 2014, 2015, and 2016. There are at least two ways that the table could be partitioned: with unbounded range partitions, or with bounded range partitions.



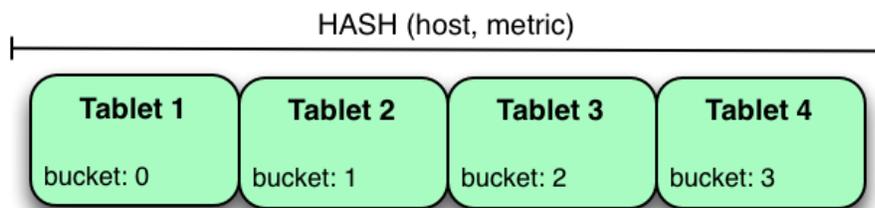
The image above shows the two ways the `metrics` table can be range partitioned on the `time` column. In the first example (in blue), the default range partition bounds are used, with splits at `2015-01-01` and `2016-01-01`. This results in three tablets: the first containing values before 2015, the second containing values in the year 2015, and the third containing values after 2016. The second example (in green) uses a range partition bound of `[(2014-01-01), (2017-01-01)]`, and splits at `2015-01-01` and `2016-01-01`. The second example could have equivalently been expressed through range partition bounds of `[(2014-01-01), (2015-01-01)]`, `[(2015-01-01), (2016-01-01)]`, and `[(2016-01-01), (2017-01-01)]`, with no splits. The first example has unbounded lower and upper range partitions, while the second example includes bounds.

Each of the range partition examples above allows time-bounded scans to prune partitions falling outside of the scan's time bound. This can greatly improve performance when there are many partitions. When writing, both examples suffer from potential hot-spotting issues. Because metrics tend to always be written at the current time, most writes will go into a single range partition.

The second example is more flexible, because it allows range partitions for future years to be added to the table. In the first example, all writes for times after `2016-01-01` will fall into the last partition, so the partition may eventually become too large for a single tablet server to handle.

Hash Partitioning

Another way of partitioning the `metrics` table is to hash partition on the `host` and `metric` columns.



In the example above, the `metrics` table is hash partitioned on the `host` and `metric` columns into four buckets. Unlike the range partitioning example earlier, this partitioning strategy will spread writes over all tablets in the table evenly, which helps overall write throughput. Scans over a specific host and metric can take advantage of partition pruning by specifying equality predicates, reducing the number of scanned tablets to one. One issue to be careful of with a pure hash partitioning strategy, is that tablets could grow indefinitely as more and more data is inserted into the table. Eventually tablets will become too big for an individual tablet server to hold.

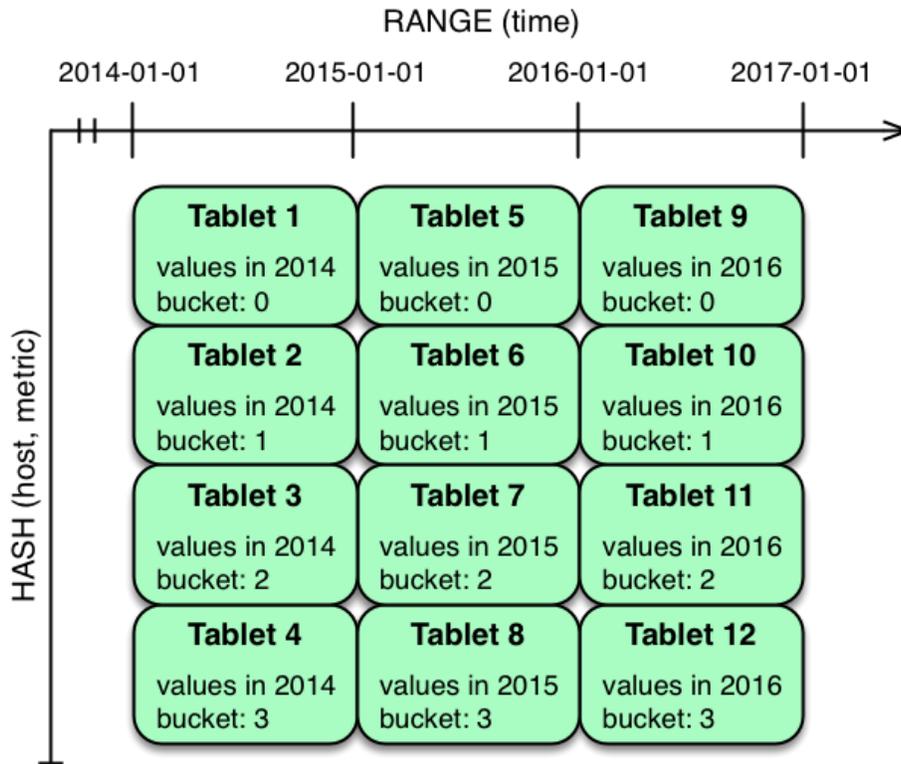
Hash and Range Partitioning

The previous examples showed how the `metrics` table could be range partitioned on the `time` column, or hash partitioned on the `host` and `metric` columns. These strategies have associated strength and weaknesses:

Table 7: Partitioning Strategies

Strategy	Writes	Reads	Tablet Growth
range(time)	☒ all writes go to latest partition	☒ time-bounded scans can be pruned	☒ new tablets can be added for future time periods
hash(host, metric)	☒ writes are spread evenly among tablets	☒ scans on specific hosts and metrics can be pruned	☒ tablets could grow too large

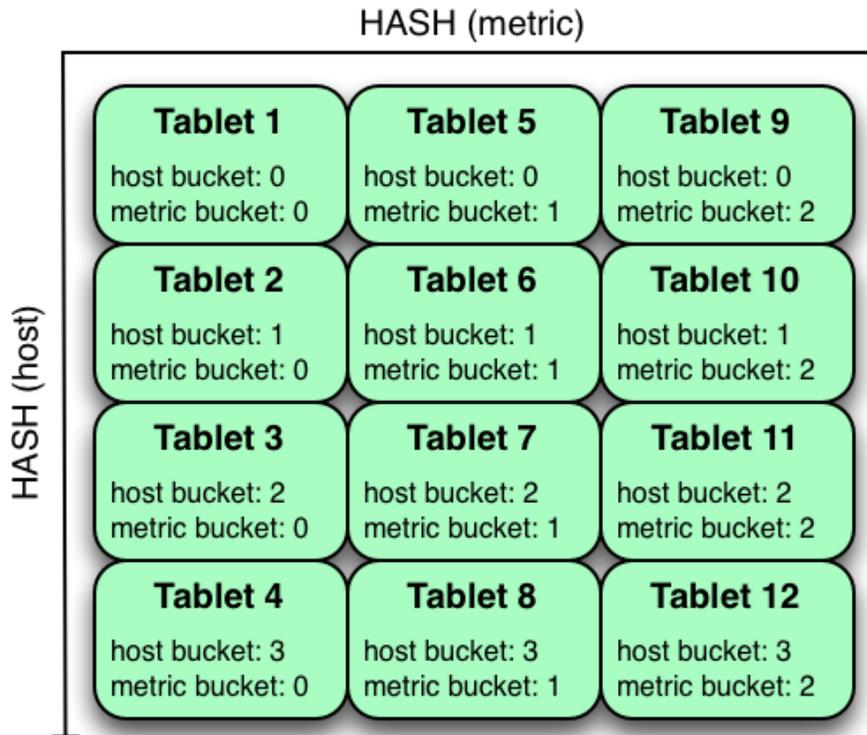
Hash partitioning is good at maximizing write throughput, while range partitioning avoids issues of unbounded tablet growth. Both strategies can take advantage of partition pruning to optimize scans in different scenarios. Using multilevel partitioning, it is possible to combine the two strategies in order to gain the benefits of both, while minimizing the drawbacks of each.



In the example above, range partitioning on the `time` column is combined with hash partitioning on the `host` and `metric` columns. This strategy can be thought of as having two dimensions of partitioning: one for the hash level and one for the range level. Writes into this table at the current time will be parallelized up to the number of hash buckets, in this case 4. Reads can take advantage of time bound **and** specific host and metric predicates to prune partitions. New range partitions can be added, which results in creating 4 additional tablets (as if a new column were added to the diagram).

Hash and Hash Partitioning

Kudu can support any number of hash partitioning levels in the same table, as long as the levels have no hashed columns in common.



In the example above, the table is hash partitioned on `host` into 4 buckets, and hash partitioned on `metric` into 3 buckets, resulting in 12 tablets. Although writes will tend to be spread among all tablets when using this strategy, it is slightly more prone to hot-spotting than when hash partitioning over multiple independent columns, since all values for an individual host or metric will always belong to a single tablet. Scans can take advantage of equality predicates on the `host` and `metric` columns separately to prune partitions.

Multiple levels of hash partitioning can also be combined with range partitioning, which logically adds another dimension of partitioning.

Schema Alterations

You can alter a table's schema in the following ways:

- Rename the table
- Rename primary key columns
- Rename, add, or drop non-primary key columns
- Add and drop range partitions

Multiple alteration steps can be combined in a single transactional operation.

Schema Design Limitations

Kudu currently has some known limitations that may factor into schema design. For a complete list, see [Apache Kudu Usage Limitations](#) on page 15.

Apache Kudu Transaction Semantics

This is a brief introduction to Kudu's transaction and consistency semantics. Kudu's core philosophy is to provide transactions with simple, strong semantics, without sacrificing performance or the ability to tune to different requirements. Kudu's transactional semantics and architecture are inspired by state-of-the-art systems such as [Spanner](#) and [Calvin](#). For an in-depth technical exposition of what is mentioned here, see the [technical report](#).

Kudu currently allows the following operations:

- **Scans** are read operations that can traverse multiple tablets and read information with different levels of consistency or correctness guarantees. Scans can also perform time-travel reads. That is, you can set a scan timestamp from the past and get back results that reflect the state of the storage engine at that point in time.
- **Write operations** are sets of rows to be inserted, updated, or deleted in the storage engine, in a single tablet with multiple replicas. Write operations do not have separate "read sets", that is, they do not scan existing data before performing the write. Each write is only concerned with the previous state of the rows that are about to change. Writes are not "committed" explicitly by the user. Instead, they are committed automatically by the system, after completion.

While Kudu is designed to eventually be fully ACID (*Atomic, Consistent, Isolated, Durable*), multi-tablet transactions have not yet been implemented. As such, the following discussion focuses on single-tablet write operations, and only briefly touches multi-tablet reads.

Single Tablet Write Operations

Kudu employs Multiversion Concurrency Control (MVCC) and the Raft consensus algorithm. Each write operation in Kudu must go through the following order of operations:

1. The tablet's leader acquires all locks for the rows that it will change.
2. The leader assigns the write a timestamp before the write is submitted for replication. This timestamp will be the write's *tag* in MVCC.
3. After a majority of replicas have acknowledged the write, the rows are changed.
4. After the changes are complete, they are made visible to concurrent writes and reads, atomically.

All replicas of a tablet observe the same process. Therefore, if a write operation is assigned timestamp n , and changes row x , a second write operation at timestamp $m > n$ is guaranteed to see the new value of x .

This strict ordering of lock acquisition and timestamp assignment is enforced to be consistent across all replicas of a tablet through consensus. Therefore, write operations are ordered with regard to clock-assigned timestamps, relative to other writes in the same tablet. In other words, writes have strict-serializable semantics.

In case of multi-row write operations, while they are *Isolated* and *Durable* in an ACID sense, they are not yet fully *Atomic*. The failure of a single write in a batch operation will not roll back the entire operation, but produce per-row errors.

Writing to Multiple Tablets

Kudu does not support transactions that span multiple tablets. However, consistent snapshot reads are possible (with caveats, as explained below). Writes from a Kudu client are optionally buffered in memory until they are flushed and sent to the tablet server. When a client's session is flushed, the rows for each tablet are batched together, and sent to the tablet server which hosts the leader replica of the tablet. Since there are no inter-tablet transactions, each of these batches represents a single, independent write operation with its own timestamp. However, the client API provides the option to impose some constraints on the assigned timestamps and on how writes to different tablets are observed by clients.

Kudu was designed to be externally consistent, that is, preserving consistency when operations span multiple tablets and even multiple data centers. In practice this means that if a write operation changes item x at tablet A , and a following write operation changes item y at tablet B , you might want to enforce that if the change to y is observed, the change to x must also be observed. There are many examples where this can be important. For example, if Kudu is storing clickstreams for further analysis, and two clicks follow each other but are stored in different tablets, subsequent clicks should be assigned subsequent timestamps so that the causal relationship between them is captured.

- **CLIENT_PROPAGATED Consistency**

Kudu's default external consistency mode is called `CLIENT_PROPAGATED`. This mode causes writes from *a single client* to be automatically externally consistent. In the clickstream scenario above, if the two clicks are submitted by different client instances, the application must manually propagate timestamps from one client to the other for the causal relationship to be captured. Timestamps between clients a and b can be propagated as follows:

Java Client

Call `AsyncKuduClient#getLastPropagatedTimestamp()` on client a , propagate the timestamp to client b , and call `AsyncKuduClient#setLastPropagatedTimestamp()` on client b .

C++ Client

Call `KuduClient::GetLatestObservedTimestamp()` on client a , propagate the timestamp to client b , and call `KuduClient::SetLatestObservedTimestamp()` on client b .

- **COMMIT_WAIT Consistency**

Kudu also has an experimental implementation of an external consistency model (used in Google's Spanner), called `COMMIT_WAIT`. `COMMIT_WAIT` works by tightly synchronizing the clocks on all machines in the cluster. Then, when a write occurs, timestamps are assigned and the results of the write are not made visible until enough time has passed so that no other machine in the cluster could possibly assign a lower timestamp to a following write.

When using this mode, the latency of writes is tightly tied to the accuracy of clocks on all the cluster hosts, and using this mode with loose clock synchronization causes writes to either take a long time to complete, or even time out.

The `COMMIT_WAIT` consistency mode may be selected as follows:

Java Client

Call `KuduSession#setExternalConsistencyMode(ExternalConsistencyMode.COMMIT_WAIT)`

C++ Client

Call `KuduSession::SetExternalConsistencyMode(COMMIT_WAIT)`



Warning:

`COMMIT_WAIT` consistency is an experimental feature. It may return incorrect results, exhibit performance issues, or negatively impact cluster stability. Its use in production environments is discouraged.

Read Operations (Scans)

Scans are read operations performed by clients that may span one or more rows across one or more tablets. When a server receives a scan request, it takes a snapshot of the MVCC state and then proceeds in one of two ways depending on the read mode selected by the user. The mode may be selected as follows:

Java Client

Call `KuduScannerBuilder#ReadMode(...)`

C++ Client

Call `KuduScanner::SetReadMode()`

The following modes are available in both clients:

READ_LATEST

This is the default read mode. The server takes a snapshot of the MVCC state and proceeds with the read immediately. Reads in this mode only yield 'Read Committed' isolation.

READ_AT_SNAPSHOT

In this read mode, scans are consistent and repeatable. A timestamp for the snapshot is selected either by the server, or set explicitly by the user through `KuduScanner::SetSnapshotMicros()`. Explicitly setting the timestamp is recommended.

The server waits until this timestamp is 'safe'; that is, until all write operations that have a lower timestamp have completed and are visible). This delay, coupled with an external consistency method, will eventually allow Kudu to have full `strict-serializable` semantics for reads and writes. However, this is still a work in progress and some [anomalies](#) are still possible. Only scans in this mode can be fault-tolerant.

Selecting between read modes requires balancing the trade-offs and making a choice that fits your workload. For instance, a reporting application that needs to scan the entire database might need to perform careful accounting operations, so that scan may need to be fault-tolerant, but probably doesn't require a to-the-microsecond up-to-date view of the database. In that case, you might choose `READ_AT_SNAPSHOT` and select a timestamp that is a few seconds in the past when the scan starts. On the other hand, a machine learning workload that is not ingesting the whole data set and is already statistical in nature might not require the scan to be repeatable, so you might choose `READ_LATEST` instead for better scan performance.



Note:

Kudu also provides replica selection API for you to choose at which replica the scan should be performed:

Java Client

```
Call KuduScannerBuilder#replicaSelection(...)
```

C++ Client

```
Call KuduScanner::SetSelection(...)
```

This API is a means to control locality and, in some cases, latency. The replica selection API has no effect on the consistency guarantees, which will hold no matter which replica is selected.

Known Issues and Limitations

There are several gaps and corner cases that currently prevent Kudu from being strictly-serializable in certain situations.

Writes

Support for `COMMIT_WAIT` is experimental and requires careful tuning of the time-synchronization protocol, such as NTP (Network Time Protocol). Its use in production environments is discouraged.

Recommendation

If external consistency is a requirement and you decide to use `COMMIT_WAIT`, the time-synchronization protocol needs to be tuned carefully. Each transaction will wait 2x the maximum clock error at the time of execution, which is usually in the 100 msec. to 1 sec. range with the default settings, maybe more. Thus, transactions would take at least 200 msec. to 2 sec. to complete when using the default settings and may even time out.

- A local server should be used as a time server. We've performed experiments using the default NTP time source available in a Google Compute Engine data center and were able to obtain a reasonable tight max error bound, usually varying between 12-17 milliseconds.
- The following parameters should be adjusted in `/etc/ntp.conf` to tighten the maximum error:
 - `server my_server.org iburst minpoll 1 maxpoll 8`
 - `tinker dispersion 500`
 - `tinker allan 0`

Reads (Scans)

- On a leader change, `READ_AT_SNAPSHOT` scans at a snapshot whose timestamp is beyond the last write, may yield non-repeatable reads (see [KUDU-1188](#)).

Recommendation

If repeatable snapshot reads are a requirement, use `READ_AT_SNAPSHOT` with a timestamp that is slightly in the past (between 2-5 seconds, ideally). This will circumvent the anomaly described above. Even when the anomaly has been addressed, back-dating the timestamp will always make scans faster, since they are unlikely to block.

- Impala scans are currently performed as `READ_LATEST` and have no consistency guarantees.
- In `AUTO_BACKGROUND_FLUSH` mode, or when using "async" flushing mechanisms, writes applied to a single client session may get reordered due to the concurrency of flushing the data to the server. This is particularly noticeable if a single row is quickly updated with different values in succession. This phenomenon affects all client API implementations. Workarounds are described in the respective API documentation for `FlushMode` or `AsyncKuduSession`. See [KUDU-1767](#).

Apache Kudu Background Maintenance Tasks

Kudu relies on running background tasks for many important maintenance activities. These tasks include flushing data from memory to disk, compacting data to improve performance, freeing up disk space, and more.

Maintenance Manager

The maintenance manager schedules and runs background tasks. At any given point in time, the maintenance manager is prioritizing the next task based on improvements needed at that moment, such as relieving memory pressure, improving read performance, or freeing up disk space. The number of worker threads dedicated to running background tasks can be controlled by setting `--maintenance_manager_num_threads`.

With Kudu 1.4, the maintenance manager features improved utilization of the configured maintenance threads. Previously, maintenance work would only be scheduled a maximum of 4 times per second, but now maintenance work will be scheduled immediately whenever any configured thread is available. Make sure that the `--maintenance_manager_num_threads` property is set to at most a 1:3 ratio for Maintenance Manager threads to the number of data directories (for spinning disks). This will improve the throughput of write-heavy workloads.

Flushing Data to Disk

Flushing data from memory to disk relieves memory pressure and can improve read performance by switching from a write-optimized, row-oriented in-memory format in the `MemRowSet`, to a read-optimized, column-oriented format on disk.

Background tasks that flush data include `FlushMRSOp` and `FlushDeltaMemStoresOp`. The metrics associated with these operations have the prefix `flush_mrs` and `flush_dms`, respectively.

With Kudu 1.4, the maintenance manager aggressively schedules flushes of in-memory data when memory consumption crosses 60 percent of the configured process-wide memory limit. The backpressure mechanism which begins to throttle client writes was also adjusted to not begin throttling until memory consumption reaches 80 percent of the configured limit. These two changes together result in improved write throughput, more consistent latency, and fewer timeouts due to memory exhaustion.

Compacting On-disk Data

Kudu constantly performs several compaction tasks in order to maintain consistent read and write performance over time.

- A merging compaction, which combines multiple `DiskRowSets` together into a single `DiskRowSet`, is run by `CompactRowSetsOp`.
- Kudu also runs two types of delta store compaction operations: `MinorDeltaCompactionOp` and `MajorDeltaCompactionOp`.

For more information on what these compaction operations do, see the [Kudu Tablet design document](#).

The metrics associated with these tasks have the prefix `compact_rs`, `delta_minor_compact_rs`, and `delta_major_compact_rs`, respectively.

Write-ahead Log Garbage Collection

Kudu maintains a write-ahead log (WAL) per tablet that is split into discrete fixed-size segments. A tablet periodically rolls the WAL to a new log segment when the active segment reaches a size threshold (configured by the `--log_segment_size_mb` property). In order to save disk space and decrease startup time, a background task called `LogGCOp` attempts to garbage-collect (GC) old WAL segments by deleting them from disk once it is determined that they are no longer needed by the local node for durability.

The metrics associated with this background task have the prefix `log_gc`.

Tablet History Garbage Collection and the Ancient History Mark

Kudu uses a multiversion concurrency control (MVCC) mechanism to ensure that snapshot scans can proceed isolated from new changes to a table. Therefore, periodically, old historical data should be garbage-collected (removed) to free up disk space. While Kudu never removes rows or data that are visible in the latest version of the data, Kudu does remove records of old changes that are no longer visible.

The specific threshold in time (in the past) beyond which historical MVCC data becomes inaccessible and is free to be deleted is called the *ancient history mark* (AHM). The AHM can be configured by setting the `--tablet_history_max_age_sec` property.

There are two background tasks that remove historical MVCC data older than the AHM:

- The one that runs the merging compaction, called `CompactRowSetsOp` (see above).
- A separate background task deletes old undo delta blocks, called `UndoDeltaBlockGCOp`. Running `UndoDeltaBlockGCOp` reduces disk space usage in all workloads, but particularly in those with a higher volume of updates or upserts. The metrics associated with this background task have the prefix `undo_delta_block`.

Kudu Scaling Guide

This document describes in detail how Kudu scales with respect to various system resources, including memory, file descriptors, and threads. See [Scaling Recommendations and Limitations](#) on page 16 for the maximum recommended parameters of a Kudu cluster. They can be used to estimate roughly the number of servers required for a given quantity of data.

Terms

We will use the following terms in this topic:

- **hot replica:** A tablet replica that is continuously receiving writes. For example, in a time series use case, tablet replicas for the most recent range partition on a time column would be continuously receiving the latest data, and would be hot replicas.
- **cold replica:** A tablet replica that is not hot, i.e. a replica that is not frequently receiving writes, for example, once every few minutes. A cold replica may be read from. For example, in a time series use case, tablet replicas for previous range partitions on a time column would not receive writes at all, or only occasionally receive late updates or additions, but may be constantly read.
- **data on disk:** The total amount of data stored on a tablet server across all disks, post-replication, post-compression, and post-encoding.

Example Workload

The sections below perform sample calculations using the following parameters:

- 200 hot replicas per tablet server
- 1600 cold replicas per tablet server
- 8TB of data on disk per tablet server (about 4.5GB/replica)
- 512MB block cache
- 40 cores per server
- limit of 32000 file descriptors per server
- a read workload with 1 frequently-scanned table with 40 columns

This workload resembles a time series use case, where the hot replicas correspond to the most recent range partition on time.

Memory

The flag `--memory_limit_hard_bytes` determines the maximum amount of memory that a Kudu tablet server may use. The amount of memory used by a tablet server scales with data size, write workload, and read concurrency. The following table provides numbers that can be used to compute a rough estimate of memory usage.

Table 8: Table 1. Tablet Server Memory Usage

Type	Multiplier	Description
Memory required per TB of data on disk	1.5GB per 1TB data on disk	Amount of memory per unit of data on disk required for basic operation of the tablet server.
Hot Replicas' MemRowSets and DeltaMemStores	minimum 128MB per hot replica	Minimum amount of data to flush per MemRowSet flush. For most use cases, updates should be rare compared to inserts, so the DeltaMemStores should be very small.
Scans	256KB per column per core for read-heavy tables	Amount of memory used by scanners, and which will be constantly needed for tables which are constantly read.
Block Cache	Fixed by <code>--block_cache_capacity_mb</code> (default 512MB)	Amount of memory reserved for use by the block cache.

Using this information for the example load gives the following breakdown of memory usage:

Table 9: Table 2. Example Tablet Server Memory Usage

Type	Amount
8TB data on disk	$8\text{TB} * 1.5\text{GB} / 1\text{TB} = 12\text{GB}$
200 hot replicas	$200 * 128\text{MB} = 25.6\text{GB}$
1 40-column, frequently-scanned table	$40 * 40 * 256\text{KB} = 409.6\text{MB}$
Block Cache	<code>--block_cache_capacity_mb=512 = 512MB</code>
Expected memory usage	38.5GB
Recommended hard limit	52GB

Using this as a rough estimate of Kudu's memory usage, select a memory limit so that the expected memory usage of Kudu is around 50-75% of the hard limit.

Verifying if a Memory Limit is sufficient

After configuring an appropriate memory limit with `--memory_limit_hard_bytes`, run a workload and monitor the Kudu tablet server process's RAM usage. The memory usage should stay around 50-75% of the hard limit, with occasional spikes above 75% but below 100%. If the tablet server runs above 75% consistently, the memory limit should be increased.

Additionally, it's also useful to monitor the logs for memory rejections, which look like:

```
Service unavailable: Soft memory limit exceeded (at 96.35% of capacity)
```

And watch the memory rejections metrics:

- `leader_memory_pressure_rejections`
- `follower_memory_pressure_rejections`
- `transaction_memory_pressure_rejections`

Occasional rejections due to memory pressure are fine and act as backpressure to clients. Clients will transparently retry operations. However, no operations should time out.

File Descriptors

Processes are allotted a maximum number of open file descriptors (also referred to as fds). If a tablet server attempts to open too many fds, it will typically crash with a message saying something like "too many open files". The following table summarizes the sources of file descriptor usage in a Kudu tablet server process:

Table 10: Table 3. Tablet Server File Descriptor Usage

Type	Multiplier	Description
File cache	Fixed by <code>--block_manager_max_open_files</code> (default 40% of process maximum)	Maximum allowed open fds reserved for use by the file cache.
Hot replicas	2 per WAL segment, 1 per WAL index	Number of fds used by hot replicas. See below for more explanation.
Cold replicas	3 per cold replica	Number of fds used per cold replica: 2 for the single WAL segment and 1 for the single WAL index.

Every replica has at least one WAL segment and at least one WAL index, and should have the same number of segments and indices; however, the number of segments and indices can be greater for a replica if one of its peer replicas is falling behind. WAL segment and index fds are closed as WALs are garbage collected.

Using this information for the example load gives the following breakdown of file descriptor usage, under the assumption that some replicas are lagging and using 10 WAL segments:

Table 11: Table 4. Example Tablet Server File Descriptor Usage

Type	Amount
file cache	40% * 32000 fds = 12800 fds
1600 cold replicas	1600 cold replicas * 3 fds / cold replica = 4800 fds
200 hot replicas	(2 / segment * 10 segments / hot replica * 200 hot replicas) + (1 / index * 10 indices / hot replica * 200 hot replicas) = 6000 fds
Total	23600 fds

So for this example, the tablet server process has about 32000 - 23600 = 8400 fds to spare.

There is typically no downside to configuring a higher file descriptor limit if approaching the currently configured limit.

Threads

Processes are allotted a maximum number of threads by the operating system, and this limit is typically difficult or impossible to change. Therefore, this section is more informational than advisory.

If a Kudu tablet server's thread count exceeds the OS limit, it will crash, usually with a message in the logs like "pthread_create failed: Resource temporarily unavailable". If the system thread count limit is exceeded, other processes on the same node may also crash.

Threads and thread pools are used all over Kudu for various purposes, but the number of threads found in nearly all of these does not scale with load or data/tablet size; instead, the number of threads is either a hard coded constant, a constant defined by a configuration parameter, or based on a static dimension (such as the number of CPU cores).

The only exception to this is the WAL append thread, one of which exists for every "hot" replica.

Note that all replicas may be considered hot at startup, so tablet servers' thread usage will generally peak when started and settle down thereafter.

Troubleshooting Apache Kudu

This guide covers basic Apache Kudu troubleshooting information. For more details, see the [official Kudu documentation for troubleshooting](#).

Issues Starting or Restarting the Master or Tablet Server

Errors During Hole Punching Test

Kudu requires hole punching capabilities in order to be efficient. Hole punching support depends upon your operation system kernel version and local filesystem implementation.

- RHEL or CentOS 6.4 or later, patched to kernel version of 2.6.32-358 or later. Unpatched RHEL or CentOS 6.4 does not include a kernel with support for hole punching.
- Ubuntu 14.04 includes version 3.13 of the Linux kernel, which supports hole punching.
- Newer versions of the ext4 and xfs filesystems support hole punching. Older versions that do not support hole punching will cause Kudu to emit an error message such as the following and to fail to start:

```
Error during hole punch test. The log block manager requires a
filesystem with hole punching support such as ext4 or xfs. On el6,
kernel version 2.6.32-358 or newer is required. To run without hole
punching (at the cost of some efficiency and scalability), reconfigure
Kudu with --block_manager=file. Refer to the Kudu documentation for more
details. Raw error message follows.
```



Note:

ext4 mountpoints may actually be backed by ext2 or ext3 formatted devices, which do not support hole punching. The hole punching test will fail when run on such filesystems. There are several different ways to determine whether an ext4 mountpoint is backed by an ext2, ext3, or ext4 formatted device; see [this Stack Exchange post](#) for details.

Without hole punching support, the log block manager is unsafe to use. It won't ever delete blocks, and will consume ever more space on disk.

If you can't use hole punching in your environment, you can still try Kudu. Enable the file block manager instead of the log block manager by adding the `--block_manager=file` flag to the commands you use to start the master and tablet servers. The file block manager does not scale as well as the log block manager.

Already present: FS layout already exists

When Kudu starts, it checks each configured data directory, expecting either for all to be initialized or for all to be empty. If a server fails to start with a log message such as the following, then this precondition check has failed.

```
Check failed: _s.ok() Bad status: Already present: Could not create new FS layout:
FSManager root is not empty: /data0/kudu/data
```

This could be because Kudu was configured with non-empty data directories on first startup, or because a previously-running, healthy Kudu process was restarted and at least one data directory was deleted or is somehow corrupted, perhaps because of a disk error. If it is the latter, refer [Changing Directory Configuration](#) on page 42.

NTP Clock Synchronization

For the master and tablet server daemons, the server's clock must be synchronized using NTP. In addition, the **maximum clock error** (not to be mistaken with the estimated error) must be below a configurable threshold. The default value is 10 seconds, but it can be set with the flag `--max_clock_sync_error_usec`.

If NTP is not installed, or if the clock is reported as unsynchronized, Kudu will not start, and will emit a message such as:

```
F0924 20:24:36.336809 14550 hybrid_clock.cc:191 Couldn't get the current time: Clock
unsynchronized. Status: Service unavailable: Error reading clock. Clock considered
unsynchronized.
```

If NTP is installed and synchronized, but the maximum clock error is too high, the user will see a message such as:

```
Sep 17, 8:13:09.873 PM FATAL hybrid_clock.cc:196 Couldn't get the current time: Clock
synchronized, but error: 11130000, is past the maximum allowable error: 10000000
```

or

```
Sep 17, 8:32:31.135 PM FATAL tablet_server_main.cc:38 Check failed: _s.ok() Bad status:
Service unavailable: Cannot initialize clock: Cannot initialize HybridClock. Clock
synchronized but error was too high (11711000 us).
```

Installing NTP

To install NTP, use the command appropriate for your operating system:

OS	Command
Debian/Ubuntu	<code>sudo apt-get install ntp</code>
RHEL/CentOS	<code>sudo yum install ntp</code>

If NTP is installed but not running, start it using one of these commands:

OS	Command
Debian/Ubuntu	<code>sudo service ntp restart</code>
RHEL/CentOS	<code>sudo /etc/init.d/ntpd restart</code>

Monitoring NTP Status

When NTP is installed, you can monitor the synchronization status by running `ntptime`. For example, a healthy system may report:

```
ntp_gettime() returns code 0 (OK)
time de24c0cf.8d5da274 Tue, Feb 6 2018 16:03:27.552, (.552210980),
maximum error 224455 us, estimated error 383 us, TAI offset 0
ntp_adjtime() returns code 0 (OK)
modes 0x0 (),
offset 1279.543 us, frequency 2.500 ppm, interval 1 s,
maximum error 224455 us, estimated error 383 us,
status 0x2001 (PLL,NANO),
time constant 10, precision 0.001 us, tolerance 500 ppm,
```

In particular, note the following most important parts of output:

- `maximum error 22455 us`: This value is well under the 10-second maximum error required by Kudu.
- `status 0x2001 (PLL,NANO)`: This indicates a healthy synchronization status.

In contrast, a system without NTP properly configured and running will output something like the following:

```
ntp_gettime() returns code 5 (ERROR)
  time de24c240.0c006000 Tue, Feb 6 2018 16:09:36.046, (.046881),
  maximum error 16000000 us, estimated error 16000000 us, TAI offset 0
ntp_adjtime() returns code 5 (ERROR)
  modes 0x0 (),
  offset 0.000 us, frequency 2.500 ppm, interval 1 s,
  maximum error 16000000 us, estimated error 16000000 us,
  status 0x40 (UNSYNC),
  time constant 10, precision 1.000 us, tolerance 500 ppm,
```

Note the UNSYNC status and the 16-second maximum error.

If more detailed information is needed, the `ntpqq` or `ntppdc` tools can be used to dump further information about which network time servers are currently acting as sources:

```
$ ntpq -nc lpeers
  remote          refid          st t when poll reach  delay  offset  jitter
-----
-108.59.2.24     130.133.1.10  2 u  13  64   1  71.743  0.373  0.016
+192.96.202.120 129.6.15.28   2 u  12  64   1  72.583 -0.426  0.028
-69.10.161.7     204.26.59.157 3 u  11  64   1  15.741  2.641  0.021
-173.255.206.154 45.56.123.24  3 u  10  64   1  43.502  0.199  0.029
-69.195.159.158 128.138.140.44 2 u   9  64   1  53.885 -0.016  0.013
*216.218.254.202 .CDMA.        1 u   6  64   1   1.475 -0.400  0.012
+129.250.35.250 249.224.99.213 2 u   7  64   1   1.342 -0.640  0.018
 45.76.244.193  216.239.35.4  2 u   6  64   1  17.380 -0.754  0.051
 69.89.207.199  212.215.1.157 2 u   5  64   1  57.796 -3.411  0.059
 171.66.97.126  .GPSs.        1 u   4  64   1   1.024 -0.374  0.018
 66.228.42.59   211.172.242.174 3 u   3  64   1  72.409  0.895  0.964
 91.189.89.198  17.253.34.125 2 u   2  64   1 135.195 -0.329  0.171
 162.210.111.4  216.218.254.202 2 u   1  64   1  28.570  0.693  0.306
 199.102.46.80  .GPS.         1 u   2  64   1  55.652 -0.039  0.019
 91.189.89.199  17.253.34.125 2 u   1  64   1 135.265 -0.413  0.037

$ ntpq -nc opeers
  remote          local          st t when poll reach  delay  offset  disp
-----
-108.59.2.24     10.17.100.238 2 u  17  64   1  71.743  0.373 187.573
+192.96.202.120 10.17.100.238 2 u  16  64   1  72.583 -0.426 187.594
-69.10.161.7     10.17.100.238 3 u  15  64   1  15.741  2.641 187.569
-173.255.206.154 10.17.100.238 3 u  14  64   1  43.502  0.199 187.580
-69.195.159.158 10.17.100.238 2 u  13  64   1  53.885 -0.016 187.561
*216.218.254.202 10.17.100.238 1 u  10  64   1   1.475 -0.400 187.543
+129.250.35.250 10.17.100.238 2 u  11  64   1   1.342 -0.640 187.588
 45.76.244.193  10.17.100.238 2 u  10  64   1  17.380 -0.754 187.596
 69.89.207.199  10.17.100.238 2 u   9  64   1  57.796 -3.411 187.541
 171.66.97.126  10.17.100.238 1 u   8  64   1   1.024 -0.374 187.578
 66.228.42.59   10.17.100.238 3 u   7  64   1  72.409  0.895 187.589
 91.189.89.198  10.17.100.238 2 u   6  64   1 135.195 -0.329 187.584
 162.210.111.4  10.17.100.238 2 u   5  64   1  28.570  0.693 187.606
 199.102.46.80  10.17.100.238 1 u   4  64   1  55.652 -0.039 187.587
 91.189.89.199  10.17.100.238 2 u   3  64   1 135.265 -0.413 187.621
```

 **Note:** Both `lpeers` and `opeers` may be helpful as `lpeers` lists `refid` and `jitter`, while `opeers` lists clock dispersion.

Using chrony for time synchronization

Some operating systems offer `chrony` as an alternative to `ntpd` for network time synchronization. Kudu has been tested most thoroughly using `ntpd` and use of `chrony` is considered experimental.

In order to use `chrony` for synchronization, `chrony.conf` must be configured with the `rtcsync` option.

NTP Configuration Best Practices

In order to provide stable time synchronization with low maximum error, follow these best NTP configuration best practices.

Always configure at least four time sources for NTP. In addition to providing redundancy in case one or more time sources becomes unavailable, The NTP protocol is designed to increase its accuracy with a diversity of sources. Even if your organization provides one or more local time servers, configuring additional remote servers is highly recommended for a robust setup.

Pick servers in your server's local geography. For example, if your servers are located in Europe, pick servers from the European NTP pool. If your servers are running in a public cloud environment, consult the cloud provider's documentation for a recommended NTP setup. Many cloud providers offer highly accurate clock synchronization as a service.

Use the `iburst` option for faster synchronization at startup. The `iburst` option instructs `ntpd` to send an initial "burst" of time queries at startup. This typically results in a faster time synchronization when a machine restarts.

An example NTP server list may appear as follows:

```
# Use my organization's internal NTP servers.
server ntp1.myorg.internal iburst
server ntp2.myorg.internal iburst
# Provide several public pool servers for
# redundancy and robustness.
server 0.pool.ntp.org iburst
server 1.pool.ntp.org iburst
server 2.pool.ntp.org iburst
server 3.pool.ntp.org iburst
```



Note: After configuring NTP, use the `ntpq` tool described above to verify that `ntpd` was able to connect to a variety of peers. If no public peers appear, it is possible that the NTP protocol is being blocked by a firewall or other network connectivity issue.

Troubleshooting NTP Stability Problems

As of Kudu 1.6.0, Kudu daemons are able to continue to operate during a brief loss of NTP synchronization. If NTP synchronization is lost for several hours, however, daemons may crash. If a daemon crashes due to NTP synchronization issues, consult the `ERROR` log for a dump of related information which may help to diagnose the issue.



Note: Kudu 1.5.0 and earlier versions were less resilient to brief NTP outages. In addition, they contained a [bug](#) which could cause Kudu to incorrectly measure the maximum error, resulting in crashes. If you experience crashes related to clock synchronization on these earlier versions of Kudu and it appears that the system's NTP configuration is correct, consider upgrading to Kudu 1.6.0 or later.



Note: NTP requires a network connection and may take a few minutes to synchronize the clock at startup. In some cases a spotty network connection may make NTP report the clock as unsynchronized. A common, though temporary, workaround for this is to restart NTP with one of the commands above.

Disk Space Usage

When using the log block manager (the default on Linux), Kudu uses [sparse files](#) to store data. A sparse file has a different apparent size than the actual amount of disk space it uses. This means that some tools may inaccurately

report the disk space used by Kudu. For example, the size listed by `ls -l` does not accurately reflect the disk space used by Kudu data files:

```
$ ls -lh /data/kudu/tserver/data
total 117M
-rw----- 1 kudu kudu 160M Mar 26 19:37 0b9807b8b17d48a6a7d5b16bf4ac4e6d.data
-rw----- 1 kudu kudu 4.4K Mar 26 19:37 0b9807b8b17d48a6a7d5b16bf4ac4e6d.metadata
-rw----- 1 kudu kudu 32M Mar 26 19:37 2f26eeacc7e04b65a009e2c9a2a8bd20.data
-rw----- 1 kudu kudu 4.3K Mar 26 19:37 2f26eeacc7e04b65a009e2c9a2a8bd20.metadata
-rw----- 1 kudu kudu 672M Mar 26 19:37 30a2dd2cd3554d8a9613f588a8d136ff.data
-rw----- 1 kudu kudu 4.4K Mar 26 19:37 30a2dd2cd3554d8a9613f588a8d136ff.metadata
-rw----- 1 kudu kudu 32M Mar 26 19:37 7434c83c5ec74ae6af5974e4909cbf82.data
-rw----- 1 kudu kudu 4.3K Mar 26 19:37 7434c83c5ec74ae6af5974e4909cbf82.metadata
-rw----- 1 kudu kudu 672M Mar 26 19:37 772d070347a04f9f8ad2ad3241440090.data
-rw----- 1 kudu kudu 4.4K Mar 26 19:37 772d070347a04f9f8ad2ad3241440090.metadata
-rw----- 1 kudu kudu 160M Mar 26 19:37 86e50a95531f46b6a79e671e6f5f4151.data
-rw----- 1 kudu kudu 4.4K Mar 26 19:37 86e50a95531f46b6a79e671e6f5f4151.metadata
-rw----- 1 kudu kudu 687 Mar 26 19:26 block_manager_instance
```

Notice that the total size reported is 117MiB, while the first file's size is listed as 160MiB. Adding the `-s` option to `ls` will cause `ls` to output the file's disk space usage.

The `du` and `df` utilities report the actual disk space usage by default.

```
$ du -h /data/kudu/tserver/data118M /data/kudu/tserver/data
```

The apparent size can be shown with the `--apparent-size` flag to `du`.

```
$ du -h --apparent-size /data/kudu/tserver/data1.7G /data/kudu/tserver/data
```

Reporting Kudu Crashes Using Breakpad

Kudu uses the [Google Breakpad](#) library to generate a minidump whenever Kudu experiences a crash. A minidump file contains important debugging information about the process that crashed, including shared libraries loaded and their versions, a list of threads running at the time of the crash, the state of the processor registers and a copy of the stack memory for each thread, and CPU and operating system version information. These minidumps are typically only a few MB in size and are generated even if core dump generation is disabled. Currently, generating minidumps is only possible on Linux deployments.

By default, Kudu stores its minidumps in a subdirectory of the configured `glog` directory called `minidumps`. This location can be customized by setting the `--minidump_path` flag. Kudu will retain only a certain number of minidumps before deleting the older ones, in an effort to avoid filling up the disk with minidump files. The maximum number of minidumps that will be retained can be controlled by setting the `--max_minidumps` gflag.

Minidumps contain information specific to the binary that created them and are therefore not useful without access to the exact binary that crashed, or a very similar binary.

Kudu developers can access the minidump tools in their development environment because they are installed as part of the Kudu thirdparty build. They can be found in the Kudu development environment under `uninstrumented/bin`. For example, `thirdparty/installed/uninstrumented/bin/minidump-2-core`.

If minidumps are enabled, it is possible to force Kudu to create a minidump without killing the process. To do that, send a `USR1` signal to the `kudu-tserver` or `kudu-master` process. For example:

```
sudo pkill -USR1 kudu-tserver
```

Viewing the minidump stack trace with the GNU Debugger

Although a minidump contains no heap information, it does contain thread and stack information. You can convert a minidump to a core file to view it with GDB.

To convert the minidump (.dmp file) to a core file:

```
minidump-2-core -o 02cb4a97-ee37-6454-73a9d9cb-590c7dde.core \
02cb4a97-ee37-6454-73a9d9cb-590c7dde.dmp
```

To view the core file with GDB (on a parcel deployment):

```
gdb /opt/cloudera/parcels/KUDU/lib/kudu/sbin-release/kudu-master \
-s /opt/cloudera/parcels/KUDU/lib/debug/usr/lib/kudu/sbin-release/kudu-master.debug \
02cb4a97-ee37-6454-73a9d9cb-590c7dde.core
```

For more information, see [Getting started with Breakpad](#) and [Linux Minidump to Core](#).

Troubleshooting Performance Issues

Kudu Tracing

The Kudu master and tablet server daemons include built-in support for tracing based on the open source [Chromium Tracing](#) framework. You can use tracing to diagnose latency issues or other problems on Kudu servers.

Accessing the Tracing Web Interface

The tracing interface is part of the embedded web server in each of the Kudu daemons, and can be accessed using a web browser. Note that while the interface has been known to work in recent versions of Google Chrome, other browsers may not work as expected.

Daemon	URL
Tablet Server	<tablet-server-1.example.com>:8050/tracing.html
Master	<master-1.example.com>:8051/tracing.html

Saving Traces

After you have collected traces, you can save these traces as JSON files by clicking **Save**. To load and analyze a saved JSON file, click **Load** and choose the file.

RPC Timeout Traces

If client applications are experiencing RPC timeouts, the Kudu tablet server WARNING level logs should contain a log entry which includes an RPC-level trace. For example:

```
W0922 00:56:52.313848 10858 inbound_call.cc:193] Call
kudu.consensus.ConsensusService.UpdateConsensus
from 192.168.1.102:43499 (request call id 3555909) took 1464ms (client timeout 1000).
W0922 00:56:52.314888 10858 inbound_call.cc:197] Trace:
0922 00:56:50.849505 (+ 0us) service_pool.cc:97] Inserting onto call queue
0922 00:56:50.849527 (+ 22us) service_pool.cc:158] Handling call
0922 00:56:50.849574 (+ 47us) raft_consensus.cc:1008] Updating replica for 2 ops
0922 00:56:50.849628 (+ 54us) raft_consensus.cc:1050] Early marking committed up to
term: 8 index: 880241
0922 00:56:50.849968 (+ 340us) raft_consensus.cc:1056] Triggering prepare for 2 ops
0922 00:56:50.850119 (+ 151us) log.cc:420] Serialized 1555 byte log entry
0922 00:56:50.850213 (+ 94us) raft_consensus.cc:1131] Marking committed up to term:
8 index: 880241
0922 00:56:50.850218 (+ 5us) raft_consensus.cc:1148] Updating last received op as
term: 8 index: 880243
0922 00:56:50.850219 (+ 1us) raft_consensus.cc:1195] Filling consensus response to
leader.
0922 00:56:50.850221 (+ 2us) raft_consensus.cc:1169] Waiting on the replicates to
finish logging
0922 00:56:52.313763 (+1463542us) raft_consensus.cc:1182] finished
```

```
0922 00:56:52.313764 (+      1us) raft_consensus.cc:1190] UpdateReplicas() finished
0922 00:56:52.313788 (+      24us) inbound_call.cc:114] Queueing success response
```

These traces can indicate which part of the request was slow. Make sure you include them when filing bug reports related to RPC latency outliers.

Kernel Stack Watchdog Traces

Each Kudu server process has a background thread called the Stack Watchdog, which monitors other threads in the server in case they are blocked for longer-than-expected periods of time. These traces can indicate operating system issues or bottle-necked storage.

When the watchdog thread identifies a case of thread blockage, it logs an entry in the `WARNING` log as follows:

```
W0921 23:51:54.306350 10912 kernel_stack_watchdog.cc:111] Thread 10937 stuck at
/data/kudu/consensus/log.cc:505 for 537ms:
Kernel stack:
[<fffffffffa00b209d>] do_get_write_access+0x29d/0x520 [jbd2]
[<fffffffffa00b2471>] jbd2_journal_get_write_access+0x31/0x50 [jbd2]
[<fffffffffa00fe6d8>] __ext4_journal_get_write_access+0x38/0x80 [ext4]
[<fffffffffa00d9b23>] ext4_reserve_inode_write+0x73/0xa0 [ext4]
[<fffffffffa00d9b9c>] ext4_mark_inode_dirty+0x4c/0x1d0 [ext4]
[<fffffffffa00d9e90>] ext4_dirty_inode+0x40/0x60 [ext4]
[<ffffffff811ac48b>] __mark_inode_dirty+0x3b/0x160
[<ffffffff8119c742>] file_update_time+0xf2/0x170
[<ffffffff8111c1e0>] __generic_file_aio_write+0x230/0x490
[<ffffffff8111c4c8>] generic_file_aio_write+0x88/0x100
[<fffffffffa00d3fb1>] ext4_file_write+0x61/0x1e0 [ext4]
[<ffffffff81180f5b>] do_sync_readv_writev+0xfb/0x140
[<ffffffff81181ee6>] do_readv_writev+0xd6/0x1f0
[<ffffffff81182046>] vfs_writev+0x46/0x60
[<ffffffff81182102>] sys_pwritev+0xa2/0xc0
[<ffffffff8100b072>] system_call_fastpath+0x16/0x1b
[<ffffffffffffffff>] 0xffffffffffffffff

User stack:
@      0x3a1ace10c4 (unknown)
@      0x1262103 (unknown)
@      0x12622d4 (unknown)
@      0x12603df (unknown)
@      0x8e7bfb (unknown)
@      0x8f478b (unknown)
@      0x8f55db (unknown)
@      0x12a7b6f (unknown)
@      0x3a1b007851 (unknown)
@      0x3a1ace894d (unknown)
@      (nil) (unknown)
```

These traces can be useful for diagnosing root-cause latency issues in Kudu especially when they are caused by underlying systems such as disk controllers or filesystems.

Memory Limits

Kudu has a hard and soft memory limit. The hard memory limit is the maximum amount a Kudu process is allowed to use, and is controlled by the `--memory_limit_hard_bytes` flag. The soft memory limit is a percentage of the hard memory limit, controlled by the flag `memory_limit_soft_percentage` and with a default value of 80%, that determines the amount of memory a process may use before it will start rejecting some write operations.

If the logs or RPC traces contain messages such as the following example, then Kudu is rejecting writes due to memory back pressure. This may result in write timeouts.

```
Service unavailable: Soft memory limit exceeded (at 96.35% of capacity)
```

There are several ways to relieve the memory pressure on Kudu:

- If the host has more memory available for Kudu, increase `--memory_limit_hard_bytes`.

- Increase the rate at which Kudu can flush writes from memory to disk by increasing the number of disks or increasing the number of maintenance manager threads `--maintenance_manager_num_threads`. Generally, the recommended ratio of maintenance manager threads to data directories is 1:3.
- Reduce the volume of writes flowing to Kudu on the application side.

Finally, in Kudu versions 1.7 and lower, check the value of the `--block_cache_capacity_mb` setting. This setting determines the maximum size of Kudu's block cache. While a higher value can help with read and write performance, setting it too high as a percentage of the `--memory_limit_hard_bytes` setting is harmful. Do not raise `--block_cache_capacity_mb` above `--memory_pressure_percentage` (default 60%) of `--memory_limit_hard_bytes`, as this will cause Kudu to flush aggressively even if write throughput is low. The recommended value for `--block_cache_capacity_mb` is below the following:

$$(50\% * \text{--memory_pressure_percentage}) * \text{--memory_limit_hard_bytes}$$

With the defaults, this means the `--block_cache_capacity_mb` should not exceed 30% of `--memory_limit_hard_bytes`.

In Kudu 1.8 and higher, servers will refuse to start if the block cache capacity exceeds the memory pressure threshold.

Block Cache Size

Kudu uses an LRU cache for recently read data. On workloads that scan a subset of the data repeatedly, raising the size of this cache can offer significant performance benefits. To increase the amount of memory dedicated to the block cache, increase the value of the `--block_cache_capacity_mb` flag. The default is 512 MiB.

Kudu provides a set of useful metrics for evaluating the performance of the block cache, which can be found on the `/metrics` endpoint of the Web UI. The following is an example set:

```
{
  "name": "block_cache_inserts",
  "value": 64
},
{
  "name": "block_cache_lookups",
  "value": 512
},
{
  "name": "block_cache_evictions",
  "value": 0
},
{
  "name": "block_cache_misses",
  "value": 96
},
{
  "name": "block_cache_misses_caching",
  "value": 64
},
{
  "name": "block_cache_hits",
  "value": 0
},
{
  "name": "block_cache_hits_caching",
  "value": 352
},
{
  "name": "block_cache_usage",
  "value": 6976
}
```

To judge the efficiency of the block cache on a tablet server, first wait until the server has been running and serving normal requests for some time, so the cache is not cold. Unless the server stores very little data or is idle, `block_cache_usage` should be equal or nearly equal to `block_cache_capacity_mb`. Once the cache has reached steady state, compare `block_cache_lookups` to `block_cache_misses_caching`. The latter metric counts the number of blocks that Kudu expected to read from cache but which weren't found in the cache. If a significant amount

of lookups result in misses on expected cache hits, and the `block_cache_evictions` metric is significant compared to `block_cache_inserts`, then raising the size of the block cache may provide a performance boost. However, the utility of the block cache is highly dependent on workload, so it's necessary to test the benefits of a larger block cache.

Heap Sampling

For advanced debugging of memory usage, or advanced debugging of memory usage, released builds of Kudu enable Heap Sampling by default. This allows Kudu developers to associate memory usage with the specific lines of code and data structures responsible. When reporting a bug related to memory usage or an apparent memory leak, heap profiling can give quantitative data to pinpoint the issue.

If heap sampling is enabled, the current sampled heap occupancy can be retrieved over HTTP by visiting `http://tablet-server.example.com:8050/pprof/heap` or `http://master.example.com:8051/pprof/heap`. The output is a machine-readable dump of the stack traces with their associated heap usage.

Rather than visiting the heap profile page directly in a web browser, it is typically more useful to use the `pprof` tool that is distributed as part of the `gperftools` open source project. For example, a developer with a local build tree can use the following command to collect the sampled heap usage and output an SVG diagram:

```
thirdparty/installed/uninstrumented/bin/pprof -svg 'http://localhost:8051/pprof/heap'  
> /tmp/heap.svg
```

The resulting SVG may be visualized in a web browser or sent to the Kudu community to help troubleshoot memory occupancy issues.

Tip: Heap samples contain only summary information about allocations and do not contain any *data* from the heap. It is safe to share heap samples in public without fear of exposing confidential or sensitive data.

Slow Name Resolution and `nscd`

For better scalability on nodes hosting many replicas, we recommend that you use `nscd` (name service cache daemon) to cache both DNS name resolution and static name resolution (via `/etc/hosts`).

When DNS lookups are slow, you will see a log message similar to the following:

```
w0926 11:19:01.339553 27231 net_util.cc:193] Time spent resolve address for  
kudu-tserver.example.com: real 4.647s user 0.000s sys 0.000s
```

`nscd` can alleviate slow name resolution by providing a cache for the most common name service requests, such as for passwords, groups, and hosts.

Refer to your operating system documentation for how to install and enable `nscd`.

Usability Issues

ClassNotFoundException: com.cloudera.kudu.hive.KuduStorageHandler

You will encounter this exception when you try to access a Kudu table using Hive. This is not a case of a missing jar, but simply that Impala stores Kudu metadata in Hive in a format that is unreadable to other tools, including Hive itself and Spark. Currently, there is no workaround for Hive users. Spark users can work around this by creating temporary tables.

Runtime Error: Could not create thread: Resource temporarily unavailable (error 11)

You will encounter this error when Kudu is unable to create more threads, usually on versions older than CDH 5.15 / Kudu 1.7. It happens on tablet servers, and is a sign that the tablet server hosts too many tablet replicas.

To fix the issue, you can raise the `nproc` ulimit as detailed in the documentation for your operating system or distribution.

However, the better solution is to reduce the number of replicas on the tablet server. This may involve rethinking the table's partitioning schema. For the recommended limits on number of replicas per tablet server, see the [known issues and scaling limitations documentation](#).

Tombstoned or STOPPED tablet replicas

You may notice some replicas on a tablet server are in a STOPPED state and remain on the server indefinitely. These replicas are tombstones. A tombstone indicates that the tablet server once held a bona fide replica of its tablet. For example, in case a tablet server goes down and its replicas are re-replicated elsewhere, if the tablet server rejoins the cluster, its replicas will become tombstones. A tombstone will remain until the table it belongs to is deleted, or a new replica of the same tablet is placed on the tablet server. A count of tombstoned replicas and details of each one are available on the `/tablets` page of the tablet server web UI. The Raft consensus algorithm that Kudu uses for replication requires tombstones for correctness in certain rare situations. They consume minimal resources and hold no data. They must not be deleted.

Corruption: checksum error on CFile block

In versions prior to Kudu 1.8.0, if the data on disk becomes corrupt, you will encounter warnings containing "Corruption: checksum error on CFile block" in the tablet server logs and client side errors when trying to scan tablets with corrupt CFile blocks. Fixing this corruption is a manual process.

To fix the issue, first identify all the affected tablets by running a checksum scan on the affected tables or tablets using the [ksck](#) tool.

```
sudo -u kudu kudu cluster ksck <master_addresses> -checksum_scan -tables=<tables>
sudo -u kudu kudu cluster ksck <master_addresses> -checksum_scan -tablets=<tablets>
```

If there is at least one replica for each tablet that does not return a corruption error, you can repair the bad copies by deleting them and forcing them to be re-replicated from the leader using the [remote_replica delete](#) tool.

```
sudo -u kudu kudu remote_replica delete <tserver_address> <tablet_id> "Cfile Corruption"
```

If all of the replica are corrupt, then some data loss has occurred. Until [KUDU-2526](#) is completed, this can happen if the corrupt replica became the leader and the existing follower replicas are replaced.

If data has been lost, you can repair the table by replacing the corrupt tablet with an empty one using the [unsafe_replace_tablet](#) tool.

```
sudo -u kudu kudu tablet unsafe_replace_tablet <master_addresses> <tablet_id>
```

From versions 1.8.0 onwards, Kudu will mark the affected replicas as failed, leading to their automatic re-replication elsewhere.

Generating a Table List

To generate a list of tables to backup using the `kudu table list` tool along with `grep` can be useful. Below is an example that will generate a list of all tables that start with `my_db`:

```
kudu table list <master_addresses> | grep "^my_db\.*" | tr '\n' ' '
```



Note: This list could be saved as a part of you backup process to be used at restore time as well.

Spark Tuning

In general the Spark jobs were designed to run with minimal tuning and configuration. You can adjust the number of executors and resources to increase parallelism and performance using Spark's [configuration options](#).

If your tables are super wide and your default memory allocation is fairly low, you may see jobs fail. To resolve this, increase the Spark executor memory. A conservative rule of thumb is 1 GiB per 50 columns.

If your Spark resources drastically outscale the Kudu cluster, then you may want to limit the number of concurrent tasks allowed to run on restore.

More Resources for Apache Kudu

The following is a list of resources that may help you to understand some of the architectural features of Apache Kudu and columnar data storage. The links further down tend toward the academic and are not required reading in order to understand how to install, use, and administer Kudu.

[Kudu Project](#)

Read the official Kudu documentation and learn how you can get involved.

[Kudu Documentation](#)

Read the official Kudu documentation, which includes more in-depth information about installation and configuration choices.

[Kudu Github Repository](#)

Examine the Kudu source code and contribute to the project.

[Kudu-Examples Github Repository](#)

View and run several Kudu code examples, as well as the Kudu Quickstart VM.

[Kudu White Paper](#)

Read draft of the white paper discussing Kudu's architecture, written by the Kudu development team.

[In Search Of An Understandable Consensus Algorithm](#), *Diego Ongaro and John Ousterhout, Stanford University. 2014.*

The original whitepaper describing the Raft consensus algorithm.

[Column-Stores vs. Row-Stores: How Different Are They Really?](#) *Abadi, Madden, Hachem. 2008.*

A discussion of the characteristics of column-based and row-based datastores and their characteristics under different workloads and schemas.

Support

Bug reports and feedback can be submitted through the [public JIRA](#), our [Cloudera Community Kudu forum](#), and a public [mailing list](#) monitored by the Kudu development team and community members. In addition, a public [Slack instance](#) is available to communicate with the team.

Cloudera Manager Metrics for Kudu

The following topics list the metrics collected by Cloudera Manager when a Kudu service is managed by Cloudera Manager. For more information about metrics in Cloudera Manager, see [Cloudera Manager Metrics](#) and [Metric Aggregation](#).

Appendix: Apache License, Version 2.0

SPDX short identifier: Apache-2.0

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims

licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution.

You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

1. You must give any other recipients of the Work or Derivative Works a copy of this License; and
2. You must cause any modified files to carry prominent notices stating that You changed the files; and
3. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
4. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions.

Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions.

Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks.

This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty.

Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability.

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability.

While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

```
Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```